VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Tvorba open-source nástroje k automatickému penetračnímu testování pro účely výuky

Creation of Open-source Tool for Automatic Penetration Testing to Be Used in Lessons

2019 Bc. Radek Svoboda

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Radek Svoboda**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 1801T064 Informační a komunikační bezpečnost

Téma: Tvorba open-source nástroje k automatickému penetračnímu testování pro účely výuky
Creation of Open-source Tool for Automatic Penetration Testing to Be Used in Lessons

Jazyk vypracování: čeština

Zásady pro vypracování:

Student provede průzkum na poli aktuálních automatických open-source penetračních nástrojů. Výstupem průzkumu bude jejich srovnání, výpis nabízených funkcí včetně bližšího popisu. Práce se bude zabývat testováním zabezpečení jak pracovních stanic, tak síťových služeb. Dle průzkumu budou vybrány vhodné aspekty zabezpečení, které budou blíže rozebrány.
V praktické části student implementuje aplikaci, která bude schopna provádět penetrační testování automaticky. Klientské části aplikace budou sloužit pro lokální testování a budou zasílat informace o stanicích na centralizované místo. Celá aplikace bude vyvíjena pro potřeby výuky a bude možné ji jednoduše rozšířit o nové funkce.

Hlavní body zadání:
1. Rešerše aktuálních open-source automatických penetračních nástrojů.
2. Popis a srovnání existujících nástrojů.
3. Návrh aplikace sloužící k výuce penetračního testování.
4. Vývoj navrhnuté aplikace.
5. Testování implementovaného řešení.

Seznam doporučené odborné literatury:

[1] ERICKSON, Jon. Hacking: umění exploitace. Brno: Zoner Press, 2005. ISBN 80-86815-21-8.
[2] HARRIS, Shon. Hacking: manuál hackera. Praha: Grada, 2008. ISBN 978-80-247-1346-5.
[3] KIM, Peter. Hacking: praktický průvodce penetračním testováním. Přeložil Jan POKORNÝ. Brno: Zoner Press, 2015. Encyklopedie Zoner Press. ISBN 978-80-7413-313-8.
[4] MCCLURE, Stuart, Joel SCAMBRAY a George KURTZ. Hacking bez záhad. Praha: Grada, 2007. ISBN 978-80-247-1502-5.
[5] WEIDMAN, Georgia. Penetration testing: a hands-on introduction to hacking. ISBN 978-1593275648.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **prof. Ing. Ivan Zelinka, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23. dubna 2019 .....

Rád bych poděkoval vedoucímu mé diplomové práce prof. Ing. Ivanu Zelinkovi, Ph.D. za poskytnuté rady a čas strávený konzultacemi.

Abstrakt

Práce se zabývá návrhem a implementací nástroje k testování adaptivních systémů pro odhalení průniku v oblasti informační bezpečnosti. Nástroj umožňuje uživateli testování těchto systémů s využitím evolučních algoritmů. Práce se skládá ze dvou částí, první část je zaměřená na problematiku automatizace penetračního testování, včetně srovnání nejpoužívanějších open-source nástrojů v této oblasti. Druhá část je věnováná návrhu a implementaci nejen penetračního nástroje, ale také adaptivního systému pro odhalení průniku na bázi strojového učení, na němž je nástroj testován.

Klíčová slova: penetrační testování, evoluční výpočetní techniky, strojové učení, adaptivní systém pro odhalení průniku

Abstract

The thesis deals with design and implementation of software tool for testing adaptive intrusion detection systems in area of information security. The tool allows the user to test these systems using evolutionary computing. The thesis consists of two parts, the first part is focused on topic of automation of penetration testing, including comparison of the most common open-source tools in this field. The second part is dedicated to design and implementation of both the penetration testing tool and the adaptive intrusion detection system based on machine learning for demonstration of usage of the tool.

Key Words: penetration testing, evolutionary computing, machine learning, adaptive intrusion detection system

Obsah

Seznam použitých zkratek a symbolů

CVE	–	Common Vulnerabilities and Exposures
CVSS	–	Common Vulnerability Scoring System
CWSS	–	Common Weakness Scoring System
DDoS	–	Distributed Denial of Service
DNS	–	Domain Name System
GUI	–	Graphical User Interface
IDS	–	Intrusion Detection System
IP	–	Internet Protocol
ISSAF	–	Information Systems Security Assessment Framework
MVC	–	Model View Controller
OSSTMM	–	Open Source Security Testing Methodology Manual
OWASP	–	Open Web Application Security Project
PTES	–	Penetration Testing Execution
REST API	–	Representational State Transfer Application Programming Interface
STRIDE	–	Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privileges
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol
VAPT	–	Vulnerability Assessment and Penetration Testing
XML	–	Extensible Markup Language

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Informační bezpečnost je, zejména v posledních letech, jedním z nejskloňovanějších pojmů takřka ve všech odvětvích lidských činností. Nezáleží na tom, zda se jedná o nadnárodní korporaci nebo pouze malý start-up, všude se setkáme s potřebou zpracovat, uložit či přenést informace a právě požadavek bezpečnosti se stal klíčovým v každé fázi zpracování informací.

Pro vytvoření kvalitního softwarového produktu je kritická nejen jeho korektnost a robustnost, ale samozřejmě také jeho bezpečnost. S bezpečností se setkáme ve všech krocích vytváření softwarového produktu, od jeho návrhu až po nasazení. Bezpečnostní vlastnosti produktu je potřeba, jako všechny ostatní, pečlivě otestovat. Manuální penetrační testování je časově velmi náročné, proto začaly vznikat nástroje, které umožňují velkou část činností automatizovat.

Nástroje určené pro automatické penetrační testování jsou zpravidla určené pro provedení předem definovaného útoku, který v zásadě probíhá stále stejně. V dnešní době začaly pronikat různé formy umělé inteligence do mnoha odvětví lidské práce a informační bezpečnost samozřejmě není výjimkou. Umělá inteligence je dnes součástí mnoha bezpečnostních produktů, uplatnění nachází zejména v oblasti síťové bezpečnosti, nicméně se nejedná o jedinou doménu její aplikace.

Využití umělé inteligence přineslo nové výzvy nejen v oblasti návrhu těchto systémů, ale také v oblasti jejich testování. Stávající penetrační nástroje jsou založeny, jak je zmíněno dříve, na určitých vzorech útoků, což bohužel v mnoha případech neumožňuje komplexní otestování systémů, které se jsou schopny vzory útoků naučit. Této problematice je diplomová práce věnována a nabízí možné řešení v podobě využití umělé inteligence v rámci automatických penetračních nástrojů.

1.1 Motivace pro použití automatických penetračních testů v dnešních systémech

V minulosti byla informační bezpečnost často chápána jako nepovinná vlastnost vytvářeného systému, jejíž implementace vyžaduje nemalou finanční a časovou investici. Z tohoto důvodu byla mnohdy bezpečnost v návrhové fázi naprosto opomíjená a byla, aspoň částečně, přidávaná až během vývoje. Kyberkriminalita je na vzestupu, dle aktuálních statistik se finanční dopad v případě úspěšného kybernetického útoku nezřídka pohybuje v miliónech dolarů [1]. Nárůst můžeme pozorovat také např. v oblasti krádeží identit, kdy za rok 2018 bylo postiženo 60 miliónů Američanů, zatímco v roce 2017 to bylo pouhých 15 miliónů [2]. Počet bezpečnostních incidentů ukázal, že zmíněný postup vývoje systémů není správný a investice do informační bezpečnosti se vyplatí.

Dnes je bezpečnost povinnou vlastnosti každého systému. S tím souvisí i změna přístupu k jejímu zakomponování do systému. Je nutné ji zahrnout již do návrhové fáze a při její implementací na ní klást stejný důraz jako na ostatní vlastnosti produktu. Nedílnou vlastnosti softwarového procesu je i testování požadovaných vlastností.

Potřeba testovat implementovanou funkcionalitu je naprosto běžná a existuje celá řada nástrojů, které pokrývají celé spektrum testů, od základních unit testů až po automatické testování uživatelského rozhraní. Rovněž bezpečnostní vlastnosti je potřeba periodicky testovat. Manuální testování má nesporné výhody, nicméně je časově náročné. Z toho důvodů začaly vznikat patřičné nástroje, obecně označované jako VAPT nástroje, které jsou schopny provést rutinní testy automaticky a je možné je snadno zakomponovat do procesu testování vyvíjeného systému.

2 Penetrační testování

Jedná se o proces testování zabezpečení organizace. Tento proces zahrnuje nejen testování hardwaru, softwaru, ale také lidí. Cílem procesu je nalézt hrozby a zranitelnosti, které může útočník reálně využít pro získání neautorizovaného přístupu do systému [3]. Výsledkem procesu je report, který obsahuje analýzu nalezených zranitelností, včetně doporučení k jejich vyřešení, aby bylo předejito bezpečnostnímu incidentu.

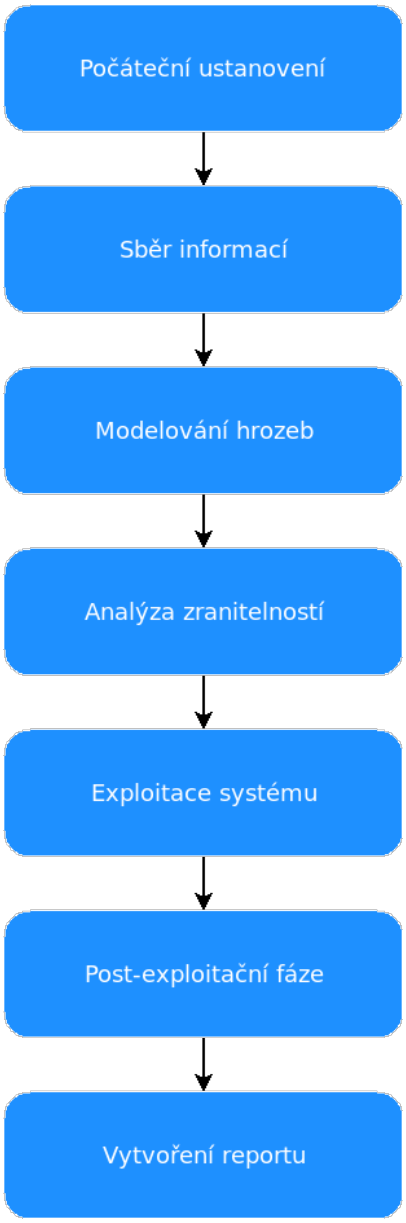
2.1 Penetrační testování a vulnerability assessments

Penetrační testování je často zaměňováno s pojmem vulnerability assessments, jedná se ale o dva rozdílné přístupy. Cílem vulnerability assessments je pouhé nalezení a zdokumentování zranitelností systému. Penetrační testování jde více do hloubky a snaží se simulovat skutečný útok zneužitím nalezených zranitelností. Tímto způsobem lze rozhodnout, které zranitelnosti jsou opravdu kritické, a které jsou tzv. falešně pozitivní, protože jejich reálné zneužití nebylo možné. Vulnerability assessments lze chápat jako jednu z části procesu penetračního testování. Oba přístupy mají své klady a zápory. Penetrační testování může být lepším indikátorem stavu zabezpečení v organizaci, nicméně se jedná o více invazivní techniku, než je tomu u vulnerability assessments. Tento fakt je potřeba zohlednit zejména u testování produkčních systémů, protože penetrační testování může narušit jejich chod, což je zpravidla nežádoucí jev.

2.2 Fáze procesu penetračního testování

Testování zabezpečení je potřeba provádět systematicky, z toho důvodu se celý proces skládá z několika fázi, které na sebe navazují. Výstupy předcházející fáze slouží jako vstup do fáze následující, podobně jako ve vodopádovém modelu u softwarového procesu [6]. Počet a pojmenování fází není pevně dáno a v publikacích se často liší, odlišnosti zpravidla vznikají různou granularitou dekompozice testovacího procesu. V praxi existuje několik standardů, které popisují metodologii testování. V závislosti na použité metodologii se liší i rozdělení na fáze. Mezi nejznámější standardy [7] patří ISSAF, OSSTMM [8] nebo PTES [4]. Kromě zmíněných obecných standardů existují i další, které se zaměřují pouze na konkrétní oblast testování, jako je např. OWASP testing guide pro testování webových aplikací.

Následující rozdělení odpovídá PTES z pohledu black-box přístupu. Zvolil jsem ji z důvodu, že se jedná se o obecnou metodologii aplikovatelnou napříč doménami s různými úrovněmi složitosti systému. Jednotlivé fáze vidíme na Obrázku 1.



Obrázek 1: Fáze penetračního testování

2.2.1 Počáteční ustanovení

V této fázi je prvním krokem stanovení cíle testování. Cíl se zpravidla odvíjí od toho, o jakou organizaci se jedná a jaký bezpečnostní incident by na ni měl největší dopad. Příkladem může být automobilová společnost, pro kterou bude zcela určitě mnohem kritičtější únik plánů prototypu připravovaného modelu vozu, zatímco DDoS útok na webové stránky tak zásadní nebude.

V této fázi je důležité stanovit také rozsah testování, tedy jaká aktiva jsme oprávnění testovat. Stejně důležité je i získání povolení k testování systémů třetích stran, pokud se v organizaci vyskytují a je nutné otestovat i je. Stejně tak by měl být stanoven časový rámec testování a limit

kam, až je možné v rámci testování zajít. Omezení jsou důležitá zejména u důležitých aktiv, u kterých by i dočasné vyřazení z provozu mělo výrazný dopad.

2.2.2 Sběr informací

Sběr informací je velmi důležitá fáze. Čím více informací se povede o cíli získat, tím snazší jsou pozdější fáze testování. Prvním krokem je získání veřejně dostupných informací [3]. Rozhodně je vhodné věnovat pozornost např. technologickému blogu, pokud jej organizace provozuje, nebo nabídkám pracovních míst. Z těchto informací je možné získat přehled o používaných technologiích, případně zde mohou být uvedeny kontaktní emailové adresy používané v rámci organizace, které mohou sloužit jako základ pro seznam možných uživatelských jmen.

Další technikou je tzv. Google hacking, pomocí kterého je možné získat např. konfigurační soubory z TFTP serverů, které jsou přístupné mimo interní síť. Další možnosti získání informací mohou být DNS záznamy, které odhalují IP adresy serverů. Ty mohou sloužit jako vstupní data v rámci aktivního scanování otevřených portů a provozovaných služeb.

2.2.3 Modelování hrozeb

Nyní stavíme na získaných informacích z předchozí fáze. Cílem je identifikace potenciálních slabých míst systému, tedy hrozeb. V případě white-box přístupu se obvykle provádí studium systému a postupná dekompozice komponent. Pro vizualizaci je možné využít např. Data flow diagramy. U každé komponenty je nutné určit hrozby, pro tento účel se často používá klasifikační schéma hrozeb STRIDE [9].

Výstupem je strom hrozeb, případně strukturovaný seznam hrozeb, který pohlíží na části systému jako na cíle útoku a popisuje kroky, které musí útočník vykonat, aby dosáhl prolomení systému. Nalezené hrozby se obvykle kategorizují. K tomuto účelu lze použít např. Common Weakness Enumeration [11]. Posledním důležitým krokem je určení závažnosti hrozeb, zde je nutné využít metriku, která je konzistentní a není subjektivní. Velmi populární metrikou je CWSS [10], která hodnotí závažnost hrozby na stupnici 0 až 100. Výsledná závažnost zohledňuje nejen riziko zranitelnosti, ale také překážky, které musí útočník překonat pro její zneužití a charakteristiky specifické pro prostředí.

2.2.4 Analýza zranitelnosti

Předmětem této fáze je určit, které hrozby mohou být reálně zneužity a přejdou ve zranitelnost. Pro určení, kterými hrozbami je vhodné se zabývat nejdříve může sloužit dříve zmíněné CWSS. Pro zjištění zranitelností se dnes velmi často používají automatické nástroje, ať už proprietární nebo open-source.

Nástroje obsahují databázi zranitelností, které jsou testovány vůči danému systému. Jejich použití celý proces velmi urychlí a ušetří mnoho práce. Tyto nástroje, ale nemají jen samé výhody. Jejich největší nevýhodou je jejich "hlučnost". Tím, že je obvykle otestováno velké množství

zranitelností v krátkém čase, se zvyšuje riziko odhalení, protože taková činnost nezůstává beze stop. Testování se zcela jistě promítne při nejmenším do přístupových logů, případně může být detekován podezřelý vzor síťového provozu pomocí IDS. Závažnost nalezených zranitelností je opět možné ohodnotit, a to pomocí CVSS [12] .

2.2.5 Exploitace systému

Jedná se jednu z posledních fází testování. Cílem je využít nalezené zranitelnosti v předchozí fázi pro proniknutí do systému. Tímto způsobem lze identifikovat nejvíce kritické zranitelnosti a vyloučit zranitelnosti, které nelze v praxi zneužít. Lze použít jak manuální přístup, tak automatické nástroje.

2.2.6 Post-exploitační fáze

Tato fáze úzce souvisí s předchozím krokem. Po proniknutí do systému nastává proces sběru informací, které mohou být hodnotné, např. se může jednat o konfigurační soubory s přihlašovacími údaji či hashe hesel k dalším službám. Sběr informací není jedinou činností, kterou lze provádět. Napadené zařízení může být použito pro přístup k dalším částem systému, které nemusí být dostupné mimo intranet. V takovém případě se obvykle celý proces testování opakuje pro další části systému. Velmi důležité je po skončení testování uvést systém, pokud je to možné, do původního stavu a nezanechat v systému např. použitý malware. Pokud tak neučiníme, vytváříme v systému další zranitelnosti, což je rozhodně nežádoucí.

Poznatky v této fázi jsou vypovídající také z pohledu analýzy rizik, protože přímo demonstrují jak velký by byl dopad na organizaci, kdyby došlo ke skutečnému bezpečnostnímu incidentu.

2.2.7 Vytvoření reportu

Výstupem závěrečné fáze je strukturovaný dokument shrnující poznatky, které byly během procesu testování zjištěny. Dokument je zpravidla rozdělen na technickou a netechnickou část. Netechnická část slouží jako shrnutí testovacího procesu a zhodnocení stavu zabezpečení s vysokou úrovní abstrakce. Součástí zhodnocení by měla být, odůvodněná, úroveň rizika vzniku bezpečnostního incidentu. Další důležitou části jsou opatření, která je potřeba učinit pro minimalizaci rizik, včetně odhadu časové náročnosti a naléhavosti jejich aplikace. Technická část obsahuje výčet zjištěných zranitelností. U každé zranitelnosti je potřeba uvést kde se vyskytuje, jaký je její dopad, jak moc je riziková a doporučení, jak daný problém vyřešit.

2.3 Přístupy k penetračnímu testování

Přístupy lze rozdělit do několika kategorií v závislosti na kritériích jako je míra informací o testovaném systému, invazivita a nebo automatizace.

Stejně jako v mnoha dalších oblastech, ani zde se nedá říct, že by některý z přístupů byl vždy lepší než ostatní. Dle mého názoru je vhodné přístupy kombinovat. Testování je možné rozdělit na dvě fáze, kdy v první z nich testujeme systém z pohledu black-box přístupu a po skončení této fáze přejdeme k white-box testování. Takový přístup sice může vyžadovat vyšší časovou investici, ale využijeme tak výhody obou přístupů.

2.3.1 Rozdělení na základě množství informací o systému

Rozlišujeme trojici kategorií na základě znalostí o systému, které jsou testerovi před započetím činnosti známy [3]. Každý přístup má své klady a zápory, volba je závislá na konkrétní situaci a hlavně cíli testování.

2.3.1.1 Black-box

U tohoto typu testování je poskytnuto naprosté minimum informací o infrastruktuře systému. V podstatě se jedná o simulaci útoku, který je veden z venčí, kdy má útočník, stejně jako v případě tohoto typu testováním informace dostupné pouze z veřejných zdrojů. Přístup je obvyklý u externího testování. Velký důraz je kladen na fázi sběru informací (viz kapitola 2.2.2). Hlavní nevýhodou tohoto přístupu je vysoká časová náročnost a také fakt, že některá slabá místa zůstanou neodhalena z důvodu absence přístupu k interním informacím. Časová náročnost plyne nejen z omezené informovanosti o testovaném prostředí, ale důvodem je často také nutnost vyhnutí se detekci probíhajícího útoku. Příkladem je zpomalení aktivního scanování síťové infrastruktury tak, aby nebyla činnost odhalena pomocí IDS. Black-box přístup obvykle zahrnuje použití automatických nástrojů, které časovou náročnost redukují.

2.3.1.2 White-box

White-box přístup je protipólem předešlého přístupu. Zde jsou k dispozici veškeré informace o infrastruktuře systému. Informace mohou zahrnovat např. zdrojové kódy vyvíjeného softwaru, používaný hardware, verze používaných síťových služeb apod. Tento přístup je obvyklý u interního testování a bývá časově méně náročný než black-box, minimálně co do fáze sběru informací. I zde je možné použít, vedle manuálního přístupu, automatické nástroje. Může se jednat jak o nástroje, které se využívají u black-box testování, tak např. o software pro statickou, resp. dynamickou, analýzu kódu apod. Zásadní nevýhodou je, že tento přístup neodráží rizika reálného útoku, protože máme k dispozici mnohem více informací, než by měl útočník. Tento přístup se hodí pro modelování situací, kdy je útok veden zevnitř organizace. Výhodou je možnost nalezení více slabých míst, než je tomu u black-box přístupu, což pramení z většího množství dostupných informací.

2.3.1.3 Gray-box

Jedná se kombinaci white-box a black-box přístupu. Před započetím testování je část, neveřejných, informací zpřístupněna, nicméně ne v takové míře jako u white-box přístupu. Výhodou je snížení časové náročnosti vzhledem k black-box testování. Gray-box přístup je výhodný zejména v případech, kdy se potřebujeme zaměřit na testování konkrétní vrstvy zabezpečení, za předpokladu, že nadřazené vrstvy byly prolomeny.

2.3.2 Rozdělení na základě míry invazivity

Míru invazivity je nutné zvážit hlavně v případě testování produkčních systémů, protože jejich vyřazení z činnosti z důvodu testování může být nepřípustné (např. lékařská zařízení) [13].

2.3.2.1 Pasivní

Jedná se o pouhé nalezení zranitelností, aniž bychom se pokoušeli o jejich zneužití, v podstatě se provádí pouze vulnerability assessment. Tento přístup nepředstavuje žádná rizika pro testovaný systém. Na druhou stranu nezjistíme, zda jsou některé z nalezených zranitelností tzv. falešně pozitivní.

2.3.2.2 Opatrné

Nalezené zranitelnosti jsou exploitovány, ale pouze v případě, že si jsme jisti, že tím systém nijak nepoškodíme. Příkladem může být testování výchozích přihlašovacích údajů pro přístup k službám.

2.3.2.3 Uvážené

Zranitelnosti jsou i v tom případě exploitovány, a to i za cenu narušení činnosti systému. Před otestováním exploitu je potřeba zvážit, jak vysoká je šance na úspěch a jaký může být dopad. Pokud je dopad nepřípustný, exploitace se neprovádí.

2.3.2.4 Agresivní

V tomto případě se provádí penetrační testování v plném rozsahu. Provádí se exploitace systému bez ohledu na důsledky v případě úspěchu. Tento přístup je zpravidla možný, pokud jsou testy prováděny ve vývojovém prostředí a produkční systémy nejsou činnosti ohroženy.

2.3.3 Rozdělení na základě míry automatizace

Rozlišujeme manuální a automatické penetrační testování. Automatickému penetračnímu testování a používaným nástrojům je věnována samostatná část práce (kapitola 3). Oba přístupy mají své opodstatnění a pro dosažení nejlepších výsledků je vhodné oba přístupy kombinovat.

2.3.3.1 Automatické testování

Tato kategorie zahrnuje ve velké míře software pro scanování, ať už se jedná o scanování síťové infrastruktury v rámci sběru informací nebo scanery zranitelností. Automatické nástroje jsou vhodné k nalezení často se vyskytujících zranitelností. Základem těchto nástrojů je databáze zranitelností, na které je systém testován. Po provedení testu se porovná odezva opět s databází a nástroj vyhodnotí, zda se zranitelnost vyskytla či nikoli. Tento přístup je časově mnohem efektivnější než manuální provádění testů.

Nevýhodou tohoto přístupu je, že zranitelnosti, které nástroj nemá v databázi otestovány nebudou. Další problém nastává ve fázi porovnání odezvy. Pokud je systém zranitelný, ale odezva neodpovídá přesně té, která je v databázi, nebude zranitelnost detekována. Opačný problém nastane ve chvíli, kdy odezva odpovídá vzoru v databázi navzdory tomu, že systém zranitelný není. Dalším problémem je fakt, že nad činnosti nástroje nemáme plnou kontrolu, což je zásadní problém zejména v případech, kdy je prioritou nízká invazivita testu [15].

2.3.3.2 Manuální testování

Manuální testování se zpravidla zaměřuje na kritická místa aplikace, která byla identifikována v rámci modelování hrozeb. Největší výhodou je fakt, že člověk je schopen porozumět business logice a aplikovat tyto znalosti v rámci testování, což je vlastnost, kterou žádný automatický nástroj nedisponuje. Ve srovnání s automatickým přístupem je manuální testování časově náročnější. Manuální přístup nicméně neznamená, že není možné použít žádné nástroje, právě naopak. Nástroje se používají, ale s tím rozdílem, že nejsou aplikovány plošně, ale tester je využívá cíleně k prozkoumání konkrétní části systému.

2.4 Shrnutí

V kapitole byl definován proces penetračního testování včetně srovnání odlišných přístupů k této činnosti. Srovnání zachycuje rozdíly v metodice a klady i zápory každého přístupu. Jak vyplývá z předešlého textu, penetrační testování je proaktivní přístup k bezpečnosti a nabízí cenné poznatky ohledné stavu zabezpečení organizace. Penetrační testování samozřejmě nemá jen samá pozitiva. Je potřeba brát v potaz několik věcí. Je zřejmé, že některé zranitelnosti nebudou odhaleny, navzdory tomu, že se v systému vyskytují. Další problém představují nové zranitelnosti, o jejichž existenci se v době provádění testů nevědělo. Z tohoto důvodu je potřeba testování provádět opakovaně.

Je potřeba mít na paměti, že penetrační testy bezpečnostní problémy nevyřeší, není to ani jejich smyslem. Efektivní je celý proces jen v případě, že se z nalezených chyb poučíme a sjednáme nápravu. Navzdory zmíněným negativům, by mělo být zařazení testování bezpečnosti do vývojového procesu naprostým minimem pro snížení rizika finanční či jiné újmy.

3 Aktuální situace v oblasti automatických penetračních nástrojů

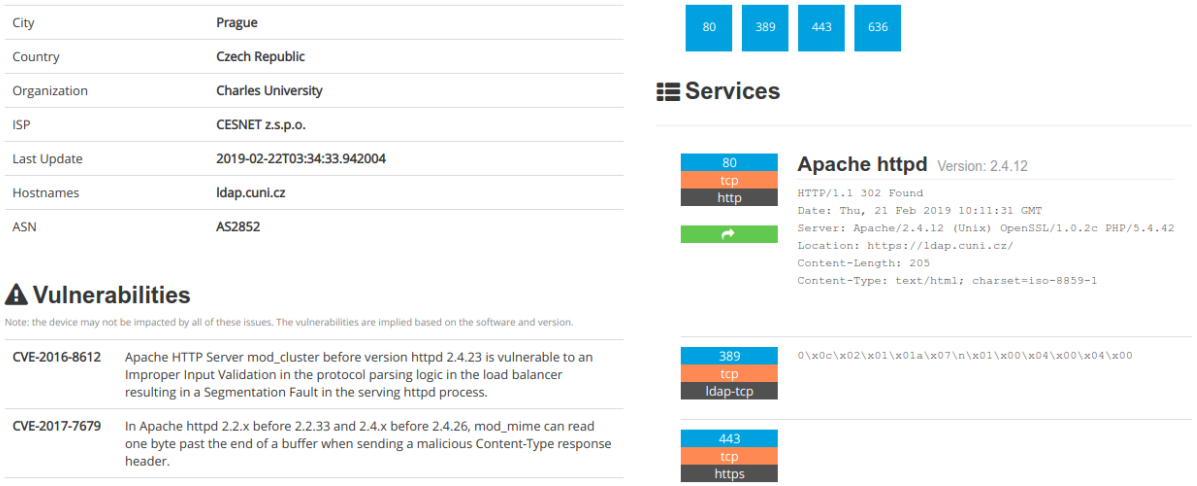
Testování zabezpečení dnešních systémů vyžaduje nemalou časovou investici. Penetrační testování je navíc specifické v tom, že je nutné jej provádět periodicky, a to i v době, kdy se nemění implementace systému. Důvodem je neustály vývoj v oblasti informační bezpečnosti, takřka denně jsou objevovány nové zranitelnosti. Čistě manuální přístup je u opakovaných činností značně neefektivní, a proto se dnes v hojné míře využívá automatických nástrojů.

Nástroje jsou využívány v testovacím procesu (viz kapitola 2.2) zejména v oblasti sběru informací, zjišťování zranitelností a jejich exploitace. Tato kapitola se zabývá problematikou automatických nástrojů ve zmíněných oblastech, vybrané nástroje jsou podrobněji popsány v kapitole 4.

3.1 Sběr informací

Rozlišujeme aktivní a pasivní nástroje. Pasivní nástroje nekomunikují přímo se servery organizace, jejíž zabezpečení testujeme, ale zaměřují se na činnost označovanou jako tzv. open source intelligence gathering, tedy získávání informací z veřejných zdrojů. Mnoho nástrojů v této kategorii cílí na získávání informací z DNS serverů. Nástroje obvykle nabízí širší možnosti, než je jen zjištění IP adres webového a mailového serveru, které umožňují i běžné linuxové nástroje jako je *nslookup* nebo *host*. Obvykle mezi ně patří např. brute force scanování poddomén ze zadaného slovníku, případně je pro tento účel využit některý z internetových vyhledávačů a poddomény jsou vyfiltrovány z výsledků hledání. Mezi často používané nástroje patří *dnsrecon* [17] nebo *dnsenum* [16].

V oblasti open source intelligence gathering jsou populární také webové služby, které shromažďují informace, obvykle získávané pasivním scanováním. Mezi nejznámější z nich patří [19] a Shodan [18]. Výhodou online služeb je, že jsou často napojeny na další zdroje dat, což jim umožňuje poskytnout více souvisejících informací. Na Obrázku 2 vidíme příklad výstupní sestavy ze služby Shodan. Kromě základních informací ve formě doménového jména a hostingu vidíme v pravém sloupci otevřené porty a informace o provozovaných službách. V případě webového serveru si můžeme všimnout, že vidíme informace z HTTP hlavičky. Obsah pole *Server* je typickým příkladem špatné konfigurace serveru, která vede k úniku informací v podobě přesných verzí používaných služeb. Tato informace je užitečná pro vyhledání informací o zranitelnostech, jejichž seznam vidíme vlevo dole. Informace z těchto služeb jsou dále využívány také v lokálně provozovaných nástrojích, příkladem je nástroj *theHarvester*, který slouží primárně k získání emailových adres a poddomén.



Obrázek 2: Příklad výstupní sestavy služby Shodan

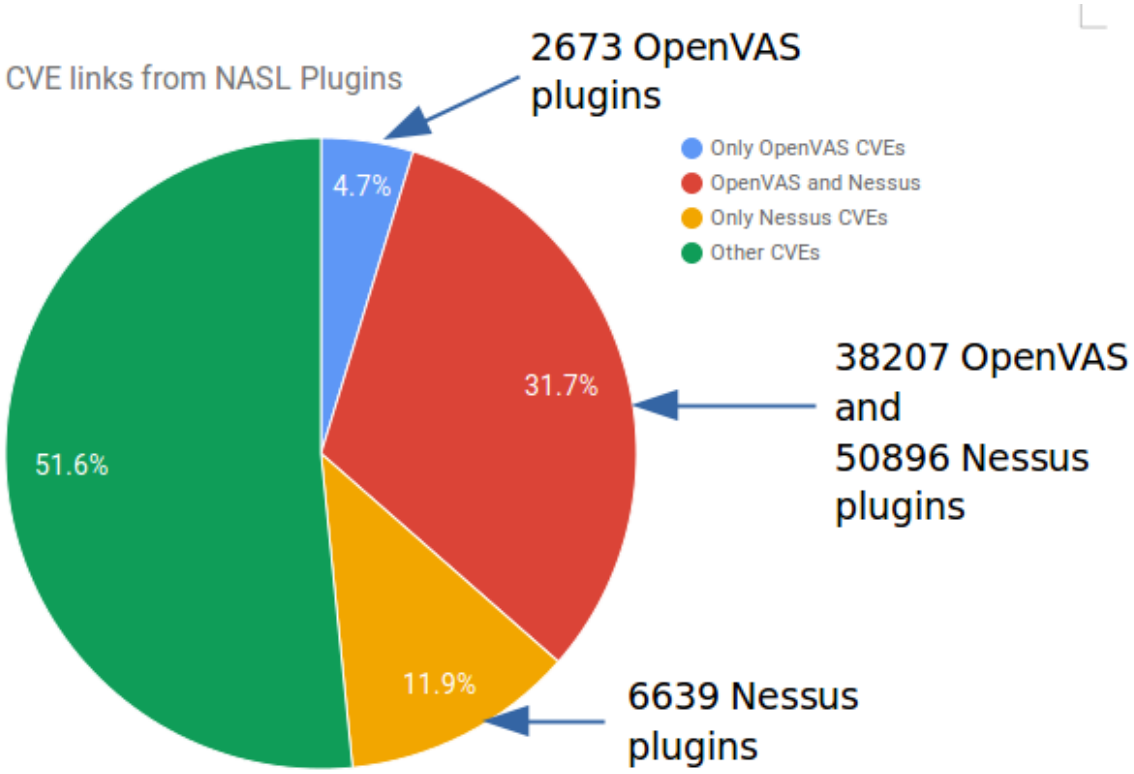
Oblast aktivního scanování zahrnuje především nástroje pro zjištění otevřených portů, spuštěných síťových služeb a jejich verzí, případně i identifikaci použitého operačního systému [5]. Často používané nástroje k těmto účelům jsou popsány v kapitole 4.

3.2 Scanování zranitelností

Na rozdíl od fáze sběru informací, kde je mnoho používaných nástrojů open-source, se v této oblasti mnohem více objevují proprietární řešení. Nástroje pro scanování zranitelností zahrnují jak všestranné síťové scanery, tak nástroje zaměřené na specifické oblasti, dnes zejména na problematiku webových aplikací.

Oblasti síťových scanerů dominují proprietární nástroje, příkladem je dobře známý Tenable Nessus [4] [22], případně Rapid7 Nexpose [23]. Jediný open-source nástroj poskytující srovnatelné možnosti je OpenVAS [21], který vychází z poslední open-source verze zmíněného Tenable Nessus. Kromě scanování zranitelností síťových služeb je možné provést také lokální scanování. V tomto případě je potřeba nakonfigurovat přístup k cílovému serveru, obvykle prostřednictvím protokolu SSH. Všechny zmíněné produkty disponují webovým rozhraním, které slouží pro konfiguraci a sledování průběhu scanování. Nástroje se neomezují na pouhé scanování zranitelností, ale i na následné činnosti, jako je klasifikace zranitelností a automatická tvorba reportu, případně i organizace nápravného procesu jako v případě nástroje Nexpose. Největší rozdíly mezi nástroji jsou v oblasti databáze zranitelností a možnosti integrace s dalšími nástroji. V oblasti integrace je na tom nejhůře OpenVAS, zásadní nevýhodou je absence REST API, které je u novějších nástrojů de facto standardem. Na Obrázku 3 vidíme porovnání databází zranitelností nástrojů OpenVAS a Nessus [24], oba nástroje používají stejný jazyk pro tvorbu zásuvných modulů, část databáze mají tedy společnou. Z grafu je patrné, že pro Nessus je k dispozici mnohem více modulů než pro OpenVAS, které navíc pokrývají více zranitelností. Metoda porovnání bohužel

není ideální, protože spoléhá na to, že v modulu je vždy uvedeno CVE zranitelnosti, kterou testuje, tudíž reálné pokrytí je pravděpodobně větší než vidíme v grafu.



Obrázek 3: Srovnání databází zranitelností nástrojů Nessus a OpenVAS, Zdroj: [24]

Open-source nástroje jsou mnohem častěji zaměřené na specifickou oblast. Dobrým příkladem jsou scanery webových aplikací. Samozřejmě existuje celá řada obecně použitelných nástrojů, které pokrývají široké spektrum zranitelnosti od špatné konfigurace webového serveru, přes cross-site scripting až po remote file inclusion. Do této kategorie řadíme např. nástroje W3af [25] nebo Nikto [26]. Na druhou stranu se zde projevuje trend typický spíše pro fázi exploitace, tedy jednoúčelové nástroje. Ty reflektují potřebu testovat konkrétní produkty, které jsou v dané oblasti široce rozšířené. Příkladem mohou být redakční systémy, kterým se nástroje velmi často věnují. Příkladem je nástroj WPScan [27] pro testování redakčního systému Wordpress, případně nástroj OWASP JoomScan [28] pro systém Joomla.

3.3 Exploitace

Trend v zásadě odpovídá situaci v oblasti scanování zranitelností. I zde jsou víceúčelové nástroje často proprietární, zatímco open-source komunita se zaměřuje na vývoj nástrojů pro konkrétní oblast použití. Nástroje lze rozdělit do dvou kategorií dle využití, a to na nástroje pro exploitaci síťových služeb a lokální využití.

Nejrozšířenějším nástrojem v této oblasti je Rapid7 Metasploit [3] [4] [29]. Kromě komerční profesionální verze je k dispozici i open-source varianta, což konkurenční nástroje jako je např. Immunity Canvas [30] nebo Secureauth Core Impact [31] nenabízejí. Metasploit je všestranný nástroj s modulární strukturou, který není omezen pouze na exploitaci, ale umožňuje i další činnosti, např. scanování síťových zařízení, vzdálený shell přístup v post-exploitační fázi a mnohé další záležitosti. Mezi jeho nesporné výhody patří možnost integrace s dalšími nástroji, a to ve dvou formách. První možnosti je import dat z nástrojů jako je již zmíněný Nessus, nebo Nmap. Druhou možnosti je použití modulu pro přímou komunikaci s těmito nástroji a spuštění scanování přímo z Metasploitu. Kromě konzolového rozhraní je pro komerční nástroje typická také přítomnost GUI, nejčastěji jako desktopová aplikace, a také podpora více platforem. Metasploit nabízí, co se týká GUI, primárně webové rozhraní, nicméně je k dispozici i open-source projekt Armitage, který přináší možnost využít desktopové rozhraní, stejně jako konkurence.

První oblasti v rámci síťových služeb, na kterou se nástroje zaměřují je získání přístupových údajů uživatelů. Pro tento účel je častá *brute force* metoda, případně extrakce ze zachyceného provozu pomocí packetových snifferů. Pro brute force metodu se nejčastěji používá trojice nástrojů: THC Hydra [32], Medusa [33] a Ncrack [34]. Hlavní odlišnosti je podpora pro různé síťové služby. THC Hydra i Medusa jsou na tom takřka stejně. THC Hydra podporuje navíc brute force přihlašovacích údajů pro Cisco zařízení. Ncrack má lepší podporu pro databáze, a to nejen pro tradiční SQL (např. MySQL nebo PostgreSQL), ale podporuje také novější NoSQL databáze, jako je např. MongoDB, Redis nebo Cassandra. Packetové sniffery pro svou činnost využívají techniku man-in-the-middle, typicky umožňují nejen zachycení provozu, ale také modifikaci obsahu a jeho replay. Nástroje jsou limitovány ve svých možnostech v závislosti na použití šifrování provozu. Mezi nejznámější nástroje patří Ettercap [36] a Mitm Proxy [35], která je zaměřena na protokoly HTTP(S) a Websocket. Za zmínku stojí i projekt Bettercap [37], který se neomezuje pouze na provoz v počítačových sítích, ale podporuje také protokol Bluetooth a manipulaci s komunikaci bezdrátových periférií komunikujících v pásmu 2,4 GHz.

Další oblastí jsou webové aplikace. Zde jsou open-source nástroje zaměřené velmi často na konkrétní hrozby, jako je např. SQL injection, file inclusion nebo obecně špatná validace vstupů, příkladem jsou dobře známé nástroje SQLMap [38] nebo OWASP ZAP [41]. Řada nástrojů se, stejně jako nástroje pro scanování zranitelností, zaměřuje na často používané redakční systémy. Jedná se např. o nástroje WordPress Exploit Framework [39] nebo XAttacker [40], který se specializuje na zranitelnosti v pluginech nejen pro Wordpress, ale také pro další redakční systémy k tvorbě blogů a e-shopů.

Nástroje pro lokální exploitaci se obvykle zaměřují na eskalaci oprávnění, vzdálený přístup a s ním spojenou extrakci informací. Tyto oblasti jsou specifické pro každý operační systém, z tohoto důvodu jsou vyvíjené nástroje zpravidla určené pro konkrétní platformu. Samozřejmě se najdou i výjimky, např. multiplatformní projekt Pupy [42], který cílí na vzdálený přístup a post-expoitační fázi. Zaměření na konkrétní platformu vedle také k hojnému využívání techniky zvané *living of the land*. To znamená, že nástroje spoléhají na prostředky, kterými daný

operační systém běžně disponuje. Pro platformu Windows jsou z tohoto důvodu nástroje často vytvořené v Powershellu. Ten navíc nabízí široké možnosti obfuskace kódu, což je důležité pro zamezení detekce nástroje antivirovým softwarem [45]. Mezi nejpokročilejší nástroje pro Windows patří Powersploit [43] a Powershell Empire [44]. Pro MacOS, resp. další unixové systémy, je situace obdobná. Pouze s tím rozdílem, že místo Powershellu se v drtivé většině případů používá jazyk Python. Pro MacOS nicméně existuje daleko méně nástrojů, než pro Windows, což přisuzuji menšímu zastoupení na trhu, zejména v oblasti pracovních stanic v podnikové sféře. Aktuálně nejpokročilejší nástroje v oblasti lokální exploitace pro MacOS jsou Bella [46] a EvilOSX [47]. Cílem extrakce informací je nezřídka získání přístupových údajů do systému. Zde vyvstává problém prolomení získaného hashe hesla, protože s uloženými hesly v otevřeném textu se dnes prakticky nesetkáme. Tento proces se zpravidla neprovádí přímo na napadeném stroji, ale prolomení probíhá na straně útočníka. V této oblasti se vývoj takřka zastavil a nové nástroje nevznikají, spíše jsou doplňovány stávající nástroje o novou funkcionalitu. Dlouhodobě nejpopulárnější nástroje jsou Hashcat [48] a John the Ripper [49].

3.4 Využití strojového učení v rámci penetračního testování

Umělá inteligence se dnes běžně uplatňuje v systémech pro detekci průniku v počítačové sítí [50] [51] nebo odhalování malwaru [52] [53]. Zmíněná problematika zahrnuje převážně klasifikační problémy. Z tohoto důvodu je ve velké míře uplatňováno strojové učení, které dosahuje výborných výsledků. Nicméně výzkum v oblasti využití strojového učení v rámci automatizace penetračního testování je v počátcích.

Nejjednodušší metodou automatizace je tzv. slepé použití VAPT nástrojů, které oscanuje systém na všechny zranitelnosti v databázi nástroje. Problémem tohoto řešení je špatná škálovatelnost, protože nástroje generují velké množství výstupních dat. Jejich analýzu a vyhodnocení musí provést expert v oblasti informační bezpečnosti, což je v rozsáhlých systémech časově náročné. V důsledku těchto problémů začaly vznikat postupy založené na rozhodovacích stromech a grafech [56], které vycházejí z modelu hrozeb [55]. Cílem bylo optimalizovat rozhodovací proces s ohledem na přednostní testování nejkritičtějších částí systému. Automatické generování plánů testování těmito postupy bylo v podstatě prvním krokem k zefektivnění testovacího procesu [54]. Tento přístup nicméně nepokrývá celý proces testování, ale omezuje se jen na fázi plánování. Zmíněné metody generují pouze statické plány procesu, což je další z problémů. Testovací proces je dynamický a jeho kroky se mohou od plánu odchýlit v závislosti na aktuálních poznatcích.

Aktuální přístupy se od těchto metod odklánějí a začínají pro zefektivnění testování používat strojové učení. Jednou z možností je jeho využití v rámci fáze sběru informací pro detekci použitých verzí síťových služeb. Metoda je založená na predikci verze na základě odpovědí serveru, využít se dají např. pole HTTP hlavičky ve formě cookie nebo e-tagu, případně výskyt specifických HTML elementů. Zjištěná verze použitého softwaru je následně využita k nalezení příslušného exploitu, pokud existuje. K tomuto účelu je už možné použít stávající nástroje, např. Metasploit. Na tomto principu je založen nástroj GyoiThon [57], který je jedním z prvních open-

source nástrojů tohoto typu. Nevýhodou takového přístupu je nutnost existence trénovacích dat v podobě signatur síťových služeb.

Stav, ke kterému automatizace penetračního testování, dle mého názoru, směřuje jsou nástroje, které se jsou schopny samostatně naučit jak testování provést. Takové nástroje by celý proces značně urychlily, protože by mohly alespoň část činností provádět bez nutnosti interakce s lidským testerem. Možností pro vytvoření takových nástrojů je využít tzv. reinforcement learning [58]. Reinforcement learning se zaměřuje na samostatné učení tzv. agentů interakcí s prostředím, ve kterém působí. Už dnes se začínají prototypy těchto nástrojů objevovat, např. nástroj Deep Exploit [59], který volně navazuje na zmíněný GyoiThon. Činnost nástroje začíná v testovacím prostředí, kde pomocí interakce s připravenými servery probíhá fáze učení. Učení spočívá opět v identifikaci zranitelností, v tuto chvíli na základě verzí použitého softwaru a následném výběru správného exploitu. Ve fázi testování reálného systému využívá agent nabyté znalosti z učící fáze. Tím je omezeno "slepé"zkoušení všech možných útoků z databáze. Agent postupuje na základě předchozích zkušeností. Takový přístup se ze zmíněných metod nejvíce podobá lidskému přístupu k testování.

3.5 Shrnutí

V této kapitole byly shrnuty trendy v oblasti automatického penetračního testování, včetně zmínění obvykle používaných nástrojů. V oblasti sběru informací je kladen velký důraz na tzv. open source intelligence gathering. Velmi populární jsou online databáze, které informace periodicky sbírají a dále poskytují testerům. Online služby jsou oblíbené i v oblasti automatických nástrojů, které v rámci své činnosti informace z nich využívají.

Scanování zranitelností a exploitace vykazují v podstatě totožný trend. V obou oblastech existují jak proprietární, tak open-source nástroje. Proprietární nástroje obvykle nabízejí široké možnosti a nezaměřují se na jedinou oblast penetračního testování. Kromě scanování, resp. exploitace, umožňují také generování podrobných reportů nebo export pro integraci s dalšími nástroji. Tyto nástroje jsou navíc snadno rozšiřitelné pomocí zásuvných modulů. Open-source komunita se zaměřuje typicky na tvorbu nástrojů, které cílí na konkrétní oblast. Obvyklé je také vytváření nástrojů kompilačního charakteru, které integrují několik jednoúčelových nástrojů do jednoho celku.

Výzkum v oblasti využití umělé inteligence, ať už v podobě strojového učení nebo jiného přístupu, v automatických penetračních nástrojích je stále v počátcích. V současné době existuje pouze několik nástrojů, které umělou inteligencí využívají, zpravidla se jedná pouze o prototypy. Proprietární nástroje v této oblasti momentálně nenajdeme, výzkum a vývoj probíhá zejména v akademické sféře a open-source komunitě. Navzdory aktuálním možnostem nástrojů má, dle mého názoru, tato problematika do budoucna obrovský potenciál, nicméně dnes není žádný nástroj na takové úrovni, aby se alespoň přiblížil kreativnímu přístupu lidí.

4 Popis a srovnání vybraných nástrojů

Kapitola se věnuje srovnání nástrojů určených především pro generování síťového provozu. Nástroje obvykle nacházejí uplatnění v rámci aktivního scanování systému, případně pro výkonnostní testování, tedy simulaci *denial of service* útoku. Srovnáním prošla trojice nástrojů: Nmap, Masscan a Hping3. Kromě těchto nástrojů samozřejmě existuje i řada dalších, nicméně jejich možnosti jsou podobné jako u zmíněné trojice nástrojů. Jedná se např. o ZMap, nebo Unicornscan. Pro otestování nástrojů je využit virtuální stroj s operačním systémem Ubuntu Server 18.04 LTS, testovány byly vždy poslední vydané verze nástrojů k datu citace.

4.1 Nmap

Nmap je open-source nástroj vytvořený v C/C++ [60]. Nástroj je primárně zaměřený na scanování síťových zařízení za účelem detekce otevřených portů, verzí síťových služeb a v neposlední řadě použitého operačního systému. Jedná se o multiplatformní software, který disponuje kromě výchozího textového rozhraní, pro použití v terminálu, také GUI. GUI je realizované prostřednictvím samostatné aplikace zvané Zenmap.

Nástroj nerozlišuje pouze otevřené a zavřené porty, ale reaguje na možnou přítomnost firewallu v komunikační trase a také na různé režimy scanování. Rozlišujeme celkově šest stavů:

- **Otevřený port**, který je dostupný a běží na něm síťová služba, je tedy možné navázat spojení.
- **Zavřený port**, který je dostupný (není filtrován na firewallu), ale neběží na něm žádná síťová služba.
- **Filtrovaný port**, u něhož není možné jednoznačně detekovat, zda je port otevřený, protože firewall zamezuje jeho dostupnosti. Zpravidla je packet zahozen, případně je odpovědí ICMP Destination unrecheable zpráva. Porty v tomto stavu znatelně zpomalují scanování, protože je potřeba provést jejich scan opakovaně, abychom se ujistili, že packety jsou skutečně zahazovány firewallem a nejedná se o výpadek v síti.
- **Nefiltrovaný port** je dostupný, ale není možné určit, zda je port otevřený nebo zavřený. Porty zařazuje do tohoto stavu pouze TCP ACK scan, který je vysvětlen dále v textu. Pro určení, zda je port otevřený je nutné použít jiný typ scanu, např. TCP SYN.
- **Otevřený/filtrovaný port**, u kterého nelze s jistotou určit, zda je port otevřený nebo filtrovaný. Typicky se tento stav vyskytuje u scanování portů UDP scanem, případně méně častými TCP NULL/FIN/XMAS režimy scanování.
- **Zavřený/filtrovaný port**, u kterého nelze s jistotou určit, zda je port zavřený nebo filtrovaný. Stav se vyskytuje pouze u IP ID scanu.

Ve vysvětlení stavu, do kterých Nmap porty klasifikuje, jsou zmíněné některé režimy scanování. Mezi základní režimy patří:

- **ICMP**, jedná se o zaslání ICMP Echo request zprávy, nescanuje porty, pouze detekuje dostupnost síťového zařízení.
- **TCP SYN**, pošle pouze TCP datagram se SYN příznakem pro započetí TCP 3-way handskake. V případě odpovědi ve formě TCP datagramu se SYN a ACK příznaky je port otevřený. Pokud je nastaven příznak RST, znamená to, že na daném portu nenaslouchá žádná síťová služba.
- **TCP SYN+ACK** spočívá v dokončení TCP 3-way handshake, na rozdíl od TCP SYN scanu. Oproti TCP SYN scanu je tato varianta pomalejší, důvodem je čekání na sestavení spojení, navíc jsou zpravidla navázaná spojení zapsána do logu. Samozřejmě i u jiných typu scanování riskujeme odhalení pomocí IDS, navzdory tomu, že do logu se pokus o spojení nezapíše.
- **TCP NULL/FIN/XMAS** jsou méně časté režimy scanování, které se liší nastavenými příznaky v datagramu. NULL scan nepoužívá žádný příznak, FIN má nastaven pouze FIN příznak a XMAS má nastavené zároveň FIN, PSH a URG příznaky. Princip scanování je shodný pro všechny tři režimy. V případě zavřeného portu je odpovědí datagram s nastaveným RST příznakem, v opačném případě není odeslána žádná odpověď. Výhodou je, že datagramy často nejsou díky své neobvyklosti zachycené IDS. Nevýhodou je, že absence odpovědi způsobuje nemožnost rozlišit mezi otevřeným a filtrovaným portem.
- **TCP ACK** je režim, který slouží pouze k zjištění, zda je port filtrovaný či nikoli. V případě nefiltrovaných portů je odpovědí datagram s nastaveným RST příznakem bez ohledu na to, zda je port otevřený nebo zavřený.
- **TCP Win** funguje na podobném principu jako TCP ACK scan. Rozdílem je analýza velikosti sliding window v přijatém datagramu. Některé systémy nastavují u otevřených portů velikost sliding window na nenulovou hodnotu i u datagramů s RST příznakem. Popsané chování je spíše výjimečné, z tohoto důvodu se režim příliš nepoužívá.
- **UDP** slouží k detekci stavu portu pro služby využívající protokol UDP. V případě zavřeného portu je odpovědí ICMP Destination unrecheable zpráva. Problematické je rozlišit otevřené porty od filtrovaných, protože v obou případech nemusí být odeslána žádná odpověď.
- **IP ID (Idle)** je varianta využívající "odraz"pro scanování. Princip je založen na inkrementaci pole ID v IP packetu. Prvním krokem je navázání spojení s tzv. zombie strojem pro zjištění hodnoty ID z IP packetu. Ve druhém kroku zašleme TCP datagram se SYN

příznakem a podvrženou IP, která odpovídá IP zombie stroje z prvního kroku cíli scanování. V případě otevřeného portu odešle cíl TCP datagram s příznaky SYN a ACK zombie stroji, ten odpoví datagramem s RST příznakem a hodnota ID je nyní o 2 větší než v prvním kroku. Pokud je port zavřený, cíl odešle pouze datagram s RST příznakem, což způsobí zvýšení hodnoty ID pouze o 1. Třetím krokem je opětovné navázání spojení se zombie strojem pro zjištění aktuální hodnoty ID. Scanování v tomto režimu je bohužel časově náročnější vzhledem ostatním variantám, na druhou stranu zůstane cíli naše IP adresa skrytá.

Kromě definovaných režimů scanování je možné vytvářet také vlastní datagramy. Pomocí vhodných přepínačů je možné nastavit libovolnou kombinaci TCP příznaků, velikost sliding window nebo sekvenční číslo. Samozřejmostí je také možnost podvrhnout IP zdroje nebo nastavit vlastní číslo portu zdroje. Datagramy mohou být doplněny o data, která jsou specifikována pomocí řetězce ASCII znaků, souboru nebo mohou být náhodně vygenerována, je nutné pouze specifikovat délku dat. Možnost úpravy datagramů je často využívána pro obejití pravidel firewallu, resp. IDS, případně pro fuzzing.

Možnost integrace s dalšími nástroji je zajištěna skrze exportování výsledku scanování. Export je umožněn textový, který odpovídá výpisu na obrazovku, případně verze po řádcích vhodná pro vyhledávání nástrojem *grep*. Další možnost exportu je XML soubor, což je výhodné pro strojové zpracování za účelem integrace s dalšími nástroji, tuto možnost využívá např. Metasploit.

Nmap umožňuje kromě pouhého scanování portů také detekci služeb na otevřených portech a verze operačního systému. Detekce síťových služeb je založena na porovnání odpovědí se vzory v databázi, Nmap je schopen rozlišit přes 2200 služeb, včetně jejich verzí. Určení operačního systému využívá rozdílů v implementaci TCP/IP stacku. Nmap nejprve pošle sekvenci různých TCP, resp. UDP, datagramů a analyzuje přijaté odpovědi. Jedním z indikátorů může být např. velikost sliding window.

Největší výhodou Nmapu vůči konkurenci je možnost vytvářet vlastní scripty s využitím Nmap script engine. Nmap navíc disponuje mnoha scripty, které jsou součástí instalace. Scripty se neomezují na pouhé scanování, ale umožňují celou řadu činnosti od fuzzingu, brute force zkoušení přihlašovacích údajů k síťovým službám až po denial of service útoky a exploitaci známých zranitelností. Pro vytváření scriptů se používá programovací jazyk Lua.

4.1.1 Příklad použití

Výpis 1 obsahuje ukázku spuštění scanu. Parametry udávají, že scan bude probíhat v režimu TCP SYN (parameter `-sS`), provede se detekce síťových služeb včetně verzí a detekce operačního systému, a to jak pomocí základní funcionality, tak pomocí vybraných scriptů (parametr `-A`). Parametr `-oN` udává výstup do souboru `ubuntu.txt` ve formátu prostého textu, poslední parametr určuje IP adresu cíle.

```
nmap -sS -A -oN ubuntu.txt 192.168.56.10
```

Výpis 1: Příklad spuštění scanování pomocí nástroje Nmap

Ve Výpisu 2 vidíme část výsledku scanování. Otevřené porty, použité síťové služby a jejich verze jsou určené naprosto přesně. Jediným problémem je nepřesná detekce operačního systému. Testovací server sice používá Linux, ale určení verze jádra je velmi nepřesné, Nmap detekuje verzi v rozpětí 3.2 až 4.8, nicméně to není správně, server používá jádro ve verzi 4.15.

```
Nmap scan report for 192.168.56.10
...
PORT    STATE SERVICE VERSION
22/tcp  open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol
    2.0)
| ssh-hostkey:
|   2048 b9:2e:1d:22:7a:34:9c:48:c6:a5:e2:ca:ea:13:99:e1 (RSA)
|   256 c6:df:86:dd:6a:ff:79:6b:8a:5e:57:fb:d5:36:96:3d (ECDSA)
|_  256 21:c0:50:6c:f7:b0:21:7f:39:d4:1f:d6:ae:c3:ea:6d (EdDSA)
53/tcp  open  domain
| dns-nsid:
|_  bind.version: 9.11.3-1ubuntu1.5-Ubuntu
80/tcp  open  http    Apache httpd 2.4.29 ((Ubuntu))
|_http-server-header: Apache/2.4.29 (Ubuntu)
|_http-title: Apache2 Ubuntu Default Page: It works
8080/tcp open http    nginx 1.14.0 (Ubuntu)
|_http-open-proxy: Proxy might be redirecting requests
|_http-server-header: nginx/1.14.0 (Ubuntu)
|_http-title: Site doesn't have a title (application/json).
MAC Address: 08:00:27:A7:C1:1F (Oracle VirtualBox virtual NIC)
Device type: general purpose
Running: Linux 3.X|4.X
OS CPE: cpe:/o:linux:linux_kernel:3 cpe:/o:linux:linux_kernel:4
OS details: Linux 3.2 - 4.8
...
```

Výpis 2: Příklad části výsledku scanování pomocí nástroje Nmap

4.1.2 Nevýhody nástroje

Nmap je kvalitní nástroj, který představuje de facto standard v oblasti aktivního scanování zařízení v síti. Nevýhod není mnoho, za hlavní považuji volbu programovacího jazyka pro tvorbu

scriptů, protože jazyk Lua není v oblasti informační bezpečnosti příliš rozšířený, dle mého názoru by mohly být scriptovací jazyky jako je Python nebo Perl lepší volbou. Poslední nevýhodou je, zejména ve starších verzích nástroje, pomalejší rychlost scanování oproti konkurenčním nástrojům. Otázkou je nakolik je rychlost důležitá, zpravidla se neprovádí testování tisíců zařízení najednou.

4.2 Masscan

Druhým testovaným nástrojem je Masscan [61], jedná se opět o open-source nástroj implementovaný čistě v jazyce C. Prioritou nástroje je co nejvyšší rychlost skenování. V případě použití speciální síťové karty (Intel 10-gbps Ethernet adapter) a ovladače je možné teoreticky přesáhnout 2 milióny poslaných packetů za sekundu. Pro práci s nástrojem je k dispozici pouze textové rozhraní. Jedná se o multiplatformní software, nicméně pro dosažení co nejvyšší rychlosti skenování se doporučuje používat Linux.

Masscan podporuje pouze dva režimy scanování, a to TCP SYN a UDP scan. Nástroj navíc nepodporuje doménová jména, vstupem může být pouze konkrétní IP adresa, případně subnet. Porty, které mají být skenovány je nutné zadat, buď výčtem nebo rozsahem, neexistuje tedy výčet výchozích portů jako v případě nástroje Nmap.

Nástroj nepodporuje vytváření vlastních datagramů jako konkurenční nástroje, jediná možnost jak ovlivnit obsah datagramu je pro protokol UDP, kde je možné specifikovat zaslaná data formou pcap souboru. Detekce operačního systému nebo alespoň verzí síťových služeb nástroj rovněž nepodporuje, získané informace jsou omezené na stav portu, tedy otevřený nebo zavřený. U detekovaných otevřených portů je možné navázat spojení a zobrazit odpověď v podobě tzv. banneru. Příkladem banneru je hlavička HTTP odpovědi webového serveru, která může být využita pro zjištění verze webového serveru z pole `Server`.

Masscan je určen primárně na skenování vysokého počtu strojů. Z tohoto důvodu nabízí nástroj pokročilejší možnosti z pohledu automatizace a možností distribuce úloh mezi více strojů než konkurence. Prvním vylepšením je, že konfiguraci nástroje je možné načíst ze souboru, což je velmi výhodné pro periodické scanování. V případě přerušení skenování je aktuální stav procesu uložen do souboru a je možné jej opětovně načíst. Posledním výrazným vylepšením je možnost tzv. shardování scanování. Shardování znamená, že scanování spustíme se stejnou konfiguraci na více strojích a každý z nich zpracuje část úlohy. Ve Výpisu 3 vidíme příklad rozdělení úlohy na tři stroje, parametr `-shard` udává, kterou část úlohy bude daný stroj zpracovávat. Integrace s jinými nástroji probíhá stejně jako v případě nástroje Nmap, výstup je možné exportovat nejen ve formě textového souboru a XML, ale přibyl i JSON.

```
masscan 0.0.0.0/0 -p0-65535 --shard 1/3
masscan 0.0.0.0/0 -p0-65535 --shard 2/3
masscan 0.0.0.0/0 -p0-65535 --shard 3/3
```

Výpis 3: Příklad shardování scanu nástrojem Masscan

4.2.1 Příklad použití

Výpis 4 demonstruje ukázkové použití nástroje Masscan. Scan je proveden pro IP adresu 192.168.2.72, nástroj měl problém provést scan přes virtuální *host-only* síťové rozhraní, proto jsem musel testovací stroj připojit do lokální sítě. TCP SYN scan prochází rozsah portů 0 až 9000 (parametr `-p`), UDP porty jsou specifikované výčtem (parametr `-udp-ports`). Posílání datagramů probíhá asynchronně, proto je nutné po odeslání počkat na odpovědi, čekání je nastaveno na 3 sekundy (parametr `-wait`). Parametr `-rate` udává kolik packetů za sekundu se odešle, čím vyšší hodnota je nastavená, tím rychleji scan probíhá. Nevýhodou je, že příliš vysoké hodnoty mohou způsobit zahlcení služby, navíc se mi často stávalo, že některé porty nebyly detekovány jako otevřené, proto doporučuji hodnoty v jednotkách tisíců packetů.

```
masscan 192.168.2.72 -p0-9000 --udp-ports 53,161 --wait 3 --rate 10000
```

Výpis 4: Příklad scanu nástrojem Masscan

Ve Výpisu 5 vidíme výsledek scanování. Masscan správně detekoval všechny otevřené porty, pro TCP i UDP. Ve srovnání s nástrojem Nmap vidíme, že výstup je mnohem méně detailní, viz. Výpis 2, kromě otevřených portů není k dispozici žádná další informace. Nástroj z nepochopitelných důvodů uvádí porty služeb využívající protokol UDP ve výpisu dvakrát.

```
Initiating SYN Stealth Scan
Scanning 1 hosts [9003 ports/host]
Discovered open port 8080/tcp on 192.168.2.72
Discovered open port 53/tcp on 192.168.2.72
Discovered open port 161/udp on 192.168.2.72
Discovered open port 161/udp on 192.168.2.72
Discovered open port 80/tcp on 192.168.2.72
Discovered open port 22/tcp on 192.168.2.72
Discovered open port 53/udp on 192.168.2.72
Discovered open port 53/udp on 192.168.2.72
```

Výpis 5: Příklad výstupu nástroje Masscan

4.2.2 Nevýhody nástroje

Masscan je nástroj, pro který je rychlost skenování prioritou číslo jedna, bohužel i na úkor další funkcionality. Hlavní nevýhodou je dle mého názoru přítomnost pouze jednoho režimu pro skenování TCP portů a naprostá absence nastavení obsahu datagramu. K obejití pravidel firewallu, resp. IDS, je nástroj absolutně nepoužitelný. Dalším problémem je špatná dokumentace nástroje, ne všechny přepínače jsou zdokumentované. V dokumentaci je pouze uvedeno, že mnoho voleb je shodných s nástrojem Nmap, bohužel už není uvedeno, o které volby se jedná.

4.3 Hping3

Hping3 je open-source nástroj vytvořený v jazyce C. Jedná se o pokročilý generátor síťového provozu se všestranným uplatněním, od fuzzingu až po simulaci DoS útoku [62]. Nástroj je možné využívat nejen prostřednictvím textového rozhraní, ale také s využitím scriptů v jazyce Tcl, podobně jako v případě nástroje Nmap.

Hping3 je především packetový generátor, který umožňuje parametrizovat většinu polí v datagramech protokolů TCP, UDP, IP a ICMP. Charakteristiky provozu nejsou v nástroji pevně vymezeny a záleží čistě na uživateli, jakou konfiguraci parametrů odesílaného provozu zvolí. Kromě konfigurace příznaků, velikosti sliding window nebo sekvenčního čísla v hlavičce TCP protokolu, která je dnes naprosto běžná u nástrojů toho typu, umožňuje Hping3 také např. ovlivnit fragmentaci IP packetů nebo podvrhnout checksum. Rozvěž je možné specifikovat data odeslaných datagramů, nástroj umí generovat náhodná data dané délky, případně je možné načíst obsah pro odeslání ze souboru.

Výsledek skenování poskytuje pouze základní informace o stavech skenovaných portů. Detekce verzí síťových služeb nebo použitého operačního systému není podporována. Nástroj navíc není schopen stáhnout ani banner, jako např. v případě nástroje Masscan. Nástroj se z tohoto důvodu hodí spíše pro testování konfigurace firewallu, resp. IDS, nebo fuzzing. Export výsledků je možný pouze prostřednictvím přesměrování výstupního proudu do souboru, což považuji za zásadní nedostatek z pohledu integrace s dalšími nástroji.

4.3.1 Příklad použití

Výpis 6 obsahuje ukázku spuštění scanování pomocí nástroje Hping3. Je provedeno scanování portů v rozsahu 0 až 9000 (parametr `-scan`) pro IP adresu `192.168.56.10`. Pro scanovací mód je implicitně použit příznak SYN u TCP datagramů, UDP scan není k dispozici. V případě nutnosti upravit parametry datagramů je možné přejít do režimu packetového generátoru, nicméně výstup již nebude ve formě přehledné tabulky.

```
hping3 --scan 0-9000 -S 192.168.56.10
```

Výpis 6: Příklad scanu nástrojem Hping3

Ve Výpisu 7 vidíme, že nástroj poskytuje základní informace ve formě nalezených otevřených portů a názvů služeb. Oproti konkurenčním nástrojům si můžeme všimnout, že výstup obsahuje také informace o přijatých odpovědích na úrovni transportní a síťové vrstvy.

```
Scanning 192.168.56.10 (192.168.56.10), port 0-9000
9001 ports to scan, use -V to see all the replies
+----+----------+---------+---+-----+-----+-----+
|port| serv name | flags |ttl| id | win | len |
+----+----------+---------+---+-----+-----+-----+
   22 ssh       : .S..A... 64    0 29200   46
   53 domain    : .S..A... 64    0 29200   46
   80 http      : .S..A... 64    0 29200   46
 8080 http-alt  : .S..A... 64    0 29200   46
All replies received. Done.
```

Výpis 7: Příklad výstupu nástroje Hping3

4.3.2 Nevýhody nástroje

Za hlavní nevýhodu nástroje považuji omezené možnosti exportu výstupních dat, integrace takových nástrojů je zpravidla značně obtížná, protože není definován žádný formát, který musí výstupní soubor dodržovat. Další nevýhodu vidím ve stručnosti výsledků skenování. Největší výhodou nástroje je možnost snadno nastavit parametry generovaného provozu a na scanovacím režimu je bohužel vidět, že se jedná pouze o možnost navíc, která není prioritou.

4.4 Shrnutí

Všechny tři testované nástroje nabízí základní funkcionalitu nutnou k provedení scanování otevřených portů. Prvním testovaným nástrojem je Nmap. Nástroj nabízí pokročilé režimy scanování, porty jsou navíc klasifikovány do více kategorií než jen otevřený a zavřený. Další výhodou je modulární struktura nástroje, která umožňuje rozšířit základní funkcionalitu vlastními scripty. Možnosti detekce operačního systému a verzí síťových služeb jsou na vysoké úrovni, ostatní nástroje v této oblasti značně zaostávají.

Masscan je zaměřen primárně na skenování velkého množství IP adres s důrazem na rychlost, bohužel se jedná o jediné výhody nástroje oproti konkurenci. Nástroj podporuje pouze základní režimy skenování a možnosti detekce použitého softwaru jsou značně omezené. Jedinou informaci poskytuje stažený banner, který ale musí analyzovat uživatel. Rozhraní vychází z nástroje Nmap, mnoho voleb je společných, bohužel dokumentace je značně nepřehledná.

Posledním testovaným nástroje je Hping3. Tento nástroj disponuje mnoha nastavitelnými parametry odesílaného provozu. Scanování portů je na nízké úrovni, konkurenční nástroje jsou dle mého názoru vhodnější volbou. Hping3 je díky své rychlosti a konfigurovatelnosti ideální volbou pro testování konfigurací firewallů, případně výkonnostní testování. Výhodou je také možnost scriptování.

Z hlediska integrace s dalšími nástroji je na tom nejhůře Hping3, protože neposkytuje strojově čitelný formát exportu, jako je např. XML nebo JSON. Všechny testované nástroje používají primárně textové rozhraní. Nmap sice nabízí GUI, nicméně to plně nepokrývá možnosti nástroje. Podporu IPv6 má pouze nástroj Nmap, je ale nutné podotknout, že scanování je mnohem pomalejší než při použití IPv4. Podpora IPv6 je dle mého názoru stěžejní pro budoucí uplatnění nástrojů tohoto typu.

Nejširší škálu možností dle mého názoru nabízí nástroj Nmap, navíc má také nejkvalitnější dokumentaci, což považuji za naprosto zásadní věc u jakéhokoli nástroje. Nejhůře hodnotím nástroj Masscan, který se snaží navázat na nástroj Nmap, bohužel kromě rychlosti scanování nepřináší žádnou přidanou hodnotu. V případě implementace podpory IPv6 by nástroj, vzhledem k své rychlosti, určitě našel širší uplatnění než je tomu nyní. Hping3 nabízí mimořádné možnosti parametrizace provozu, nabízí ale jen základní možnosti scanování, navíc nepodporuje strojově čitelný export výsledků. Navzdory zmíněným nedostatkům hodnotím nástroj kladně, protože jeho použití je velmi intuitivní a má přehlednou dokumentaci. Přehledné shrnutí rozdílů mezi nástroji nabízí Tabulka 1. Přehlednost dokumentace je hodnocena známkou z intervalu 1 - 5 (1 = nejlepší, 5 = nejhorší).

Tabulka 1: Rozdíly mezi nástroji

Vlastnosti	Nástroje		
	Nmap	Masscan	Hping3
TCP SYN scan	Ano	Ano	Ano
UDP scan	Ano	Ano	Ne
Pokročilé režimy scanování	Ano	Ne	Ne
Parametrizace provozu	Ano	Ne	Ano
Detekce OS	Ano	Ne	Ne
Detekce verzí síť. služeb	Ano	Ne	Ne
Strojově čitelný export (XML, JSON)	Ano	Ano	Ne
Načtení konfigurace ze souboru	Ne	Ano	Ne
Podpora IPv6	Ano	Ne	Ne
GUI	Zenmap	-	-
Jazyk pro tvorbu modulů	Lua	-	Tcl
Přehlednost dokumentace	1	3	1

5 Evoluční výpočetní techniky a strojové učení

Cílem kapitoly je poskytnout základní informace o algoritmech z oblasti evolučních výpočetních technik a strojového učení. Tyto algoritmy jsou stěžejní pro činnost komponent testovacího prostředí.

5.1 Evoluční výpočetní techniky

Evoluční výpočetní techniky řadíme do třídy optimalizačních algoritmů. Cílem těchto algoritmů je nalezení takové numerické kombinace vstupních argumentů účelové funkce, při kterých je dosaženo jejího globálního optima. Největší uplatnění nalézají evoluční algoritmy v oblasti optimalizace funkcí, které se vyznačují vysokým výskytem lokálních optim. Důvodem je zahrnutí stochastické složky do jednotlivých kroků algoritmu, která umožňuje překlenout lokální optimum a pokračovat v prohledávání prostoru možných řešení. Tento přístup představuje obrovskou výhodu oproti gradientním metodám, které jsou zaměřeny zpravidla na lokální optimalizaci a konvergují tedy pouze k lokálnímu optimu, stochastická složka je navíc zahrnuta pouze ve fázi výběru počátečního řešení při opakovaném spouštění. Další nespornou výhodou je schopnost řešit tzv. black-box problémy, u kterých není účelová funkce analyticky definována, mnohem efektivněji než čistě stochastické algoritmy. Navíc není výstup algoritmu omezen na jediné řešení, ale je jich nalezeno několik, což je výhodné zejména u víceúčelové optimalizace. Za nevýhodu evolučních algoritmů je považován dlouhý čas přechodu ze sub-optimálního řešení k optimálnímu. Tato nevýhoda se dá snadno řešit kombinaci více algoritmů, kdy je nejprve prostor řešení prohledáván pomocí evolučního algoritmu a nejlepší nalezená řešení následně slouží jako počáteční body pro metody lokálního prohledávání. Ty zahrnují nejen gradientní metody, ale i další metody se stochastickou složkou, jako je např. simulované žíhání [66].

Existuje celá řada evolučních algoritmů, které se liší v prvé řadě přístupem k optimalizaci, ale také tím, jak detailně prostor možných řešení prohledávají. Přístupům k optimalizaci je velmi často inspirací příroda, algoritmy tedy vycházejí z Darwinovy a Mendelovy teorie evoluce, pohybu mravenců, hejnové inteligence a mnohých dalších jevů. Zjednodušeně demonstruje činnost evolučních algoritmů Algoritmus 1, pseudokód je zaměřen především na algoritmy vycházející z Darwinovy teorie, nicméně princip dalších přístupů je velmi podobný, jen použitá terminologie se zpravidla liší. Algoritmy pracují s populacemi jedinců, jedinec představuje konkrétní kombinaci parametrů, tedy nalezené řešení. Prvotní populace je volena náhodně, vytváření dalších populací je již dáno deterministickým postupem. Celý proces evoluce se opakuje dokud není splněná ukončovací podmínka, tou může být např. počet iterací nebo velikost rozdílu nejlepších nalezených řešení v poslední a předposlední populaci. Každému jedinci je přiřazena hodnota, tzv. fitness, která určuje kvalitu nalezeného řešení. Každá etapa evoluce začíná volbou rodičů, zpravidla dle fitness, ale není to pravidlem. Následně jsou vytvoření potomci aplikací operátorů daného algoritmu na rodiče. Operátory se v rámci různých algoritmů liší, v zásadě se ale jedná o křížení rodičů a následnou mutaci vytvořených potomků. Způsob jak křížení a mutace

probíhají může být ovlivněn jak reprezentací jedinců, tak i řešeným problémem. Následně jsou vybrání nejlepší jedinci z řad rodičů a potomků. Ti tvoří novou populaci, která starou nahrazuje. Celý proces se následně opakuje znova.

Vstup : Parametry algoritmu
Výstup: Nalezené řešení

```
1 Vygenerování počáteční populace jedinců
2 Ohodnocení jedinců v počáteční populaci
3 while ukončovací podmínka není splněna do
4 |   Volba rodičů dle fitness
5 |   Vytvoření potomků aplikací operátorů na zvolené rodiče
6 |   Výběr nejlepších jedinců z rodičů i potomků
7 |   Nahrazení původní populace novou
8 end
```

Algoritmus 1: Zjednodušený pohled na činnost evolučních algoritmů

Algoritmů postavených na tomto principu existuje celá řada a obecně platí tzv. *No free lunch theorem (NFLT)*, který říká, že pro každý problém je vhodný jiný algoritmus. Algoritmy se liší exploračními a exploitačními schopnostmi. Explorace je zaměřená na prohledání co největší oblasti prostoru možných řešení i za cenu nižší úrovně detailnosti prohledávání, zatímco exploitace cílí na prohledání pouze malé oblasti, ale mnohem detailněji. Schopnosti algoritmu jsou ovlivněny nejen principem fungování, ale také nastavením parametrů.

5.1.1 Principy vybraných algoritmů

V nástroji je implementováno několik vybraných evolučních algoritmů, tato podkapitola se věnuje jejich stručnému popisu, pro více informací viz [64].

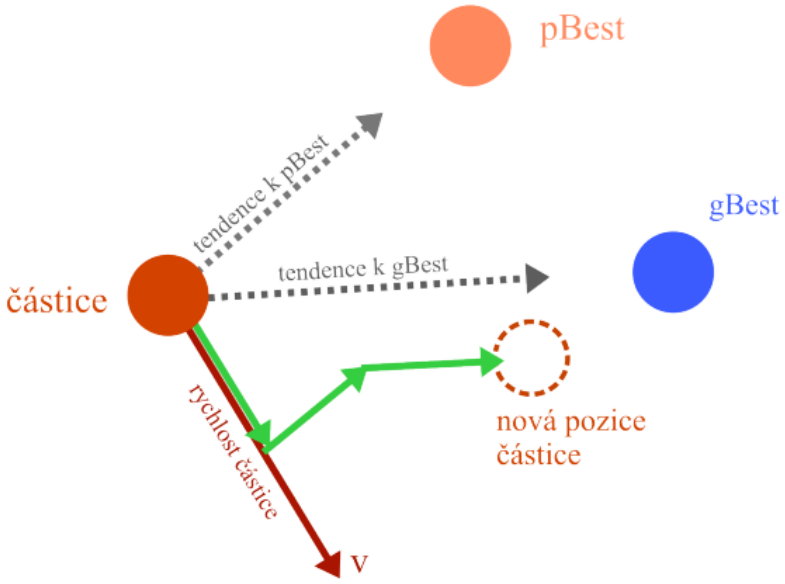
5.1.1.1 Self-Organizing Migrating Algorithm (SOMA)

SOMA je hejnový algoritmus založený na principu soutěže a kooperace jedinců, princip vychází z hledání zdroje potravy, autorem je I. Zelinka [63]. Algoritmus pracuje s populacemi jedinců, kteří se v migračních kolech pohybují v prostoru možných řešení. Ve fázi kooperace si jedinci sdělí své pozice v prostoru společně s kvalitou nalezeného řešení, které odpovídá aktuální pozici. Průběh fáze soutěžení se odvíjí od zvolené strategie, v textu je popsána strategie *All-To-One*, nicméně existují mnohé další. Strategie *All-To-One* spočívá v pohybu jedinců směrem k pozici jedince, který nalezl aktuálně nejlepší řešení. Pohyb jedinců probíhá ve skocích, po provedení daného počtu skoků se jedinec přesune na pozici nejlepšího nalezeného řešení v rámci těchto skoků. Stochastická složka je zajištěna perturbačním vektorem, který "ruší"pohyb jedince, čímž je zajištěno, že jedinec se nepohybuje přímo k cíli, ale jeho pohyb je vychylován. Po dokončení fáze kooperace a soutěže je migrační kolo dokončeno. Prohledávání je ovlivněno hyperparametry algoritmu, které udávají délku skoku, dále jak daleko od pozice vedoucího jedince aktuální jedinec

skončí a také jak moc je pohyb perturbován. Mírou perturbace lze zvýhodnit explorační nebo naopak exploitační schopnosti algoritmu.

5.1.1.2 Particle Swarm Optimization (PSO)

Jedná se o algoritmus využívající hejnovou inteligenci, autory jsou J. Kennedy a R. Eberhart [65]. Algoritmus je inspirován pohybem hejn ptáků, resp. ryb, v prostoru. Algoritmus pracuje opět s populací jedinců, každý jedinec má přiřazenou kromě své pozice a fitness, také rychlost. Pozice jedince v prostoru reprezentuje nalezené řešení, každý jedinec si pamatuje svou nejlepší pozici a zná také nejlepší nalezenou pozici v rámci celé populace. Při pohybu jedince se uplatňuje tendence přesunu na svou nejlepší pozici a zároveň na nejlepší pozici v rámci populace, míra ovlivnění je volena náhodně, čímž je zajištěna stochastická složka pro lepší explorační schopnosti. Překlenuta vzdálenost je ovlivněna rychlosti jedince. Rychlost se v každém kroku mění, proto je potřeba stanovit její limit, aby nedošlo k překročení mezí prohledávané oblasti.

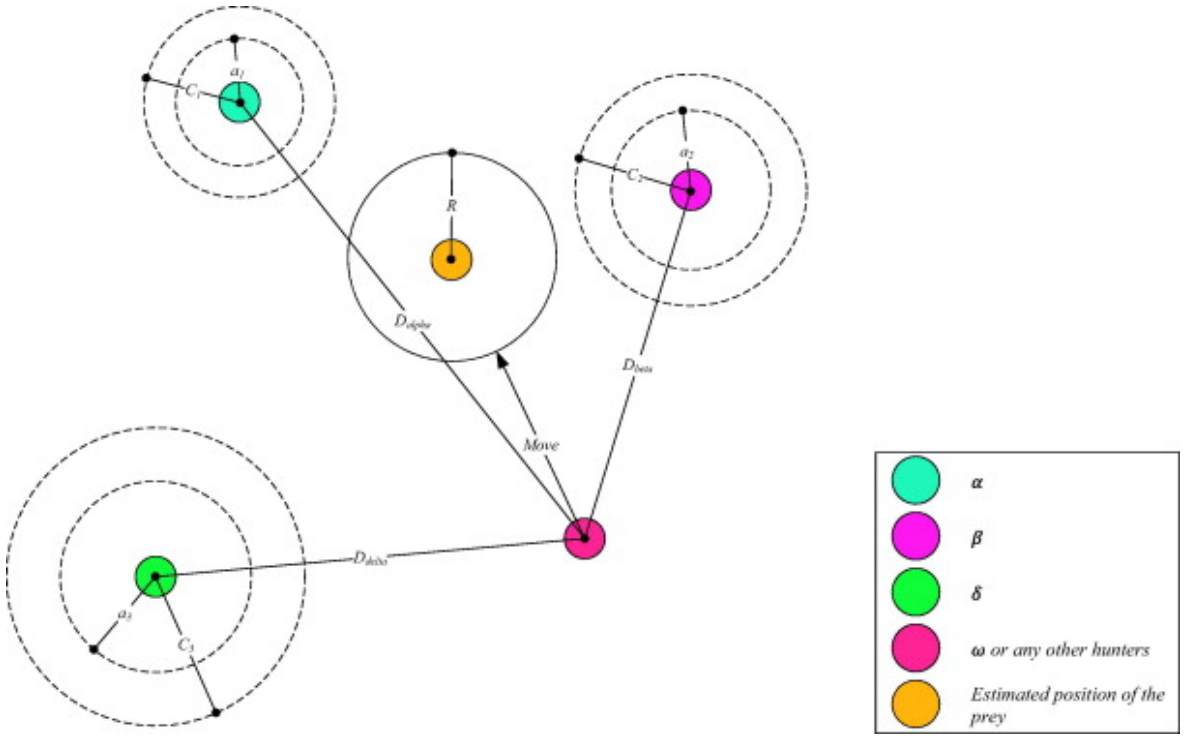


Obrázek 4: Ovlivnění pohybu částice algoritmu PSO, Zdroj: [66]

5.1.1.3 Grey Wolf Optimization (GWO)

GWO je opět hejnový algoritmus, který vychází z pohybu vlků při lovu. Algoritmus využívá hierarchické struktury smečky, kdy rozlišujeme tři vedoucí jedince - alfa, beta a delta. Alfa představuje nejlepší nalezené řešení, beta druhé nejlepší a delta třetí nejlepší [67]. Zbytek jedinců spadá do kategorie omega. Pozice kořisti je odvozená z pozic jedinců alfa, beta a delta, jak je vidět na Obrázku 5. Pohyb vlků ovlivňuje pozice vedoucích jedinců. Vlci se snaží při lovu kořist obklíčit a tento princip je využit i zde. Za kořist je považován střed prostoru mezi vedoucími

jedinci, ostatní jedinci se pohybují směrem k němu, čímž je simulováno obklíčení. Vzdálenost od kořisti je volena náhodně, tím je zajištěna stochastická složka algoritmu.



Obrázek 5: Ovlivnění pohybu jedince algoritmu GWO, Zdroj: [67]

5.1.1.4 Differential Evolution (DE)

Diferenciální evoluci publikovali v roce 1995 R. Storn a K. Price, algoritmus vychází z genetického algoritmu, používá i stejnou terminologii, nicméně přístupem k evoluci se oba algoritmy liší [68]. DE pracuje s generacemi jedinců, na které aplikuje operátor mutace a křížení. Mutace spočívá ve vytvoření šumového vektoru ze tří různých jedinců, kteří jsou k rodiči vybráni. Ve výchozí variantě se šumový vektor vypočítá přičtení váženého rozdílu prvních dvou vybraných jedinců k jedinci třetímu. Proces křížení spočívá v sekvenčním výběru dvojic složek šumového vektoru a rodiče. Pro každou složku je vygenerováno náhodné číslo. Do zkušebního vektoru, tzn. potomka, se umístí ty složky šumového vektoru, pro které bylo vygenerováno menší číslo než je práh křížení, na ostatní pozice jsou umístěny složky vektoru rodiče. Následně se zkušební vektor ohodnotí účelovou funkcí, pokud je fitness lepší než fitness rodiče, nahradí potomek rodiče v populaci. Tento proces se opakuje pro každou generaci.

5.1.1.5 Simulated Annealing (SA)

Simulované žíhání je lokální optimalizační metoda, nejedná se ale o evoluční techniku, algoritmus je založen na sousedství. Inspirací je proces žíhání tuhého tělesa [69] [66]. Prvním krokem

je náhodná volba počátečního bodu. Následně je v každém kroku vybráno `n` sousedních bodů z jeho okolí, v případě spojité funkce může být okolí dáno např. normálním rozdělením $N(\mu, \sigma^2)$. Každý bod je ohodnocen účelovou funkcí, lepší řešení je automaticky přijato. Přijetí horšího řešení je rovněž možné, ale pravděpodobnost je ovlivněna aktuální teplotou. Na počátku prohledávání je teplota nastavena na maximum a postupně dochází k její redukci, způsob redukce určuje plán chlazení. S teplotou klesá i pravděpodobnost přijetí horšího řešení, postupně tedy klesají explorační schopnosti algoritmu ve prospěch exploitačních. Celý proces končí dosažením minimální teploty.

5.2 Strojové učení

Strojové učení je oblast zahrnující algoritmy pro získání vzorů ze vstupních dat. Nalezené vzory jsou reprezentovány modelem. Algoritmy pracují s množinou instancí, které jsou popsány atributy. Atributy mohou být jak nominální, tak numerické. Jednotlivým instancím může být přiřazena také třída, který označuje do jaké kategorie je instance zařazena. Algoritmy dělíme dle použití do dvou kategorií, první je učení s učitelem, druhou učení bez učitele. Učení s učitelem zahrnuje klasifikační a regresní algoritmy. Smyslem klasifikačních algoritmů je vytvoření modelu na základě dostupných dat, kde má každá instance určenou třídu, pro následnou predikci třídy nových, dosud neviděných, instancí. Regrese funguje na totožném principu, jediným rozdílem je, že algoritmy nepredikují třídu, tedy nominální hodnotu, ale číslo. Příkladem klasifikačního problému je dobře známý XOR problém [70], regresní úloha je např. predikce cen nemovitostí. Učení bez učitele zahrnuje především shlukovací algoritmy, které rozdělují instance do určitého počtu skupin (shluků), aniž by byly instance opatřeny známou třídou. Následná analýza spočívá v nalezení společných vlastností instancí v jednotlivých shlucích. Vyvíjený systém využívá klasifikační algoritmy, řešený problém spadá do oblasti binární klasifikace, z toho důvodu se tato část práce zaměřuje především na ně. Pro více informací viz [71].

5.2.1 Naïve Bayes

Naïve Bayes je klasifikační algoritmus založený na Bayesově větě. Cílem klasifikace je určit, pro kterou ze tříd je tzv. posteriorní pravděpodobnost nejvyšší. To je dáno vztahem (1), kde C je náhodná proměnná reprezentující třídu, a $\overline{X} = (a_1, ..., a_d)$ je vektor atributů instance. Naivní přístup spočívá v předpokladu, že jednotlivé atributy jsou nezávislé, to obvykle u reálných dat nenastává, což degraduje přesnost algoritmu. Nicméně jak je uvedeno ve vztahu (1), pro aproximaci lze předpoklad zanedbat [71]. Algoritmus je pro svou jednoduchost a hlavně mnohem vyšší rychlost než ostatní algoritmy často používán v dynamických systémech. Pro možnost

rychlého přepočtu modelu, je algoritmus hojně využíván pro úlohy jako je např. detekce spamu.

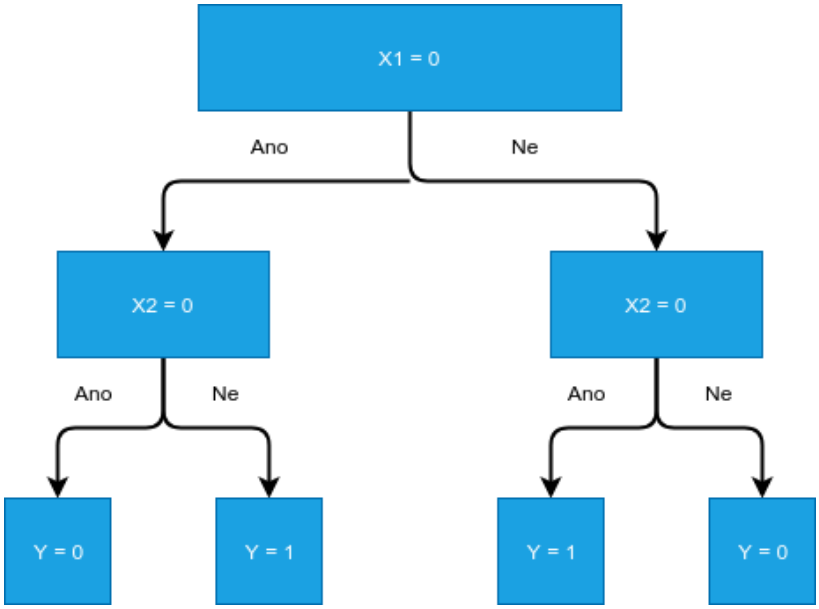
$$\begin{aligned} P(C = c|\overline{X} = (a_1, ..., a_d)) &= P(C = c|x_1 = a_1, ..., x_d = a_d) = \\ &= \frac{P(x_1 = a_1, ..., x_d = a_d|C = c) \cdot P(C = c)}{P(x_1 = a_1, ..., x_d = a_d)} \approx \\ &\approx P(C = c) \prod_{j=1}^{d} P(x_j = a_j|C = c) \end{aligned} \tag{1}$$

5.2.2 K-Nearest Neighbors

Tento klasifikační algoritmus je založen na sousedství. Princip algoritmu spočívá v tom, že v prvním kroku se vypočte vzdálenost klasifikované instance od všech ostatních bodů v datové sadě, metrika vzdálenosti není pevně dána, může se jednat např. o Euklidovskou vzdálenost, volba záleží na konkrétním problému. Druhým krokem je nalézt `k` instancí, které jsou klasifikované instanci nejblíže. Instance je následně zařazena do majoritní třídy v množině nejbližších instancí. Počet sousedních instancí je důležitým parametrem, který ovlivňuje přesnost klasifikace [72].

5.2.3 Decision Tree

Algoritmus používá pro predikci rozhodovací strom. Strom je konstruován typicky metodou depth-first [73]. Nejprve definujeme podmínku, která rozdělí množinu instancí do dvou disjunktivních podmnožin. Tato podmínka odpovídá kořeni stromu, kde levý podstrom pracuje s podmnožinou instancí, které podmínku splňují, pravý podstrom pracuje s těmi, které ne, samozřejmě je možné volit podstromy opačně. Stejným způsobem se dále děli množiny na dvojice podmnožin v dalších uzlech stromu. Konstrukce stromu končí ve chvíli, kdy jsou v jednotlivých podmnožinách pouze instance se stejnou třídou. Listy stromu nesou informaci do jaké třídy bude instance, která daný list průchodem stromu dosáhne, zařazena. Typicky probíhá dělení na podmnožiny na základě jediného atributu v jednom uzlu, to jak je množiny nejvhodnější rozdělit se obvykle určuje na základě entropie nebo gini indexu. Na Obrázku 6 vidíme ukázku úplného rozhodovacího stromu pro XOR problém.



Obrázek 6: Příklad rozhodovacího stromu pro XOR problém

5.2.4 Random Forest

Random forest je velmi populární ensemble metoda vycházející z algoritmu Decison tree. Problémem rozhodovacích stromů je snadné dosažení tzv. přeučení (overfitting), což znamená, že model není schopen správně generalizovat, ale spíše si zapamatoval trénovací data. Právě na tento problém reaguje Random forest, který zavádí stochastickou složku. Rozlišujeme dva přístupy, a to *sample bagging* a *feature bagging*. První zmíněná metoda nejprve vybere náhodnou podmnožinu instancí z celé datové sady, výběr probíhá s opakováním. Následně je vytvořen rozhodovací strom pouze pro tuto podmnožinu, stejný postup se opakuje pro požadovaný počet stromů. Druhá varianta je častější a spočívá v tom, že se náhodně volí podmnožina atributů a při konstrukci stromu se pracuje jen s nimi, místo toho, aby se použily všechny. Stochastická složka zaručuje větší diverzitu modelu, což omezuje zmíněnou tendenci rozhodovacích stromů k přeučení. Nevýhodou algoritmu je, že konstrukce většího počtu stromu, i následná predikce, je časově náročná, proto se algoritmus v základní verzi nehodí pro real time systémy.

5.2.5 Algoritmy využivající boosting

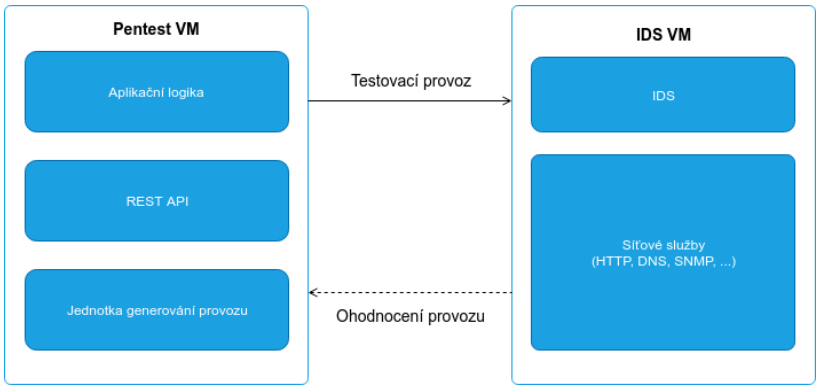
Boosting je sekvenční ensemble metoda, která je založena na kombinací několika jednodušších klasifikátoru [75]. Nejčastěji se jedná o rozhodovací stromy s velmi nízkou hloubkou. V první fázi algoritmu je všem instancím přiřazena stejná váha. Následně se vytvoří rozhodovací strom a na základě klasifikace trénovacích dat se určí, které instance byly klasifikovány správně, a které špatně. Na základě chybovosti modelu je mu přiřazena váha, která určuje jeho důležitost vzhledem k ostatním modelům. Dalším krokem je přizpůsobení vah instancí tak, aby byla věnována zvýšená pozornost špatně klasifikovaným instancím. To je zajištěno zahrnutím váhy do výpočtu

gini indexu, resp. entropie, při dělení na podmnožiny v rámci konstrukce rozhodovacího stromu. Tento proces je opakován každou iterací. Výsledkem je model, jenž se skládá z několika jednoduchých modelů. Predikce pak probíhá kombinací výstupu dílčích modelů, samozřejmě s ohledem na jejich váhy. Na tomto principu funguje např. algoritmus Adaboost. Nevýhodou metody je sekvenční přístup, takže není možné výpočet modelu provést paralelně, což vede ke zvýšení časové náročnosti, proto není ani tento algoritmus vhodný pro použití v real time systémech.

6 Návrh testovacího prostředí

Vyvíjený penetrační nástroj cílí na doménu adaptivních IDS. Adaptivní systémy nejsou založené na statických pravidlech, které jsou do systému vloženy administrátorem, ale z dostupných dat vytvoří tato pravidla sám systém. Primárním účelem nástroje je generovat takový provoz, který je možné považovat za za škodlivý, ale IDS jej momentálně není schopen správně detekovat. Cíleným generováním tohoto provozu dojde k tomu, že adaptivní IDS se jej naučí správně klasifikovat, čímž je zvýšená pravděpodobnost včasné detekce případné reálné hrozby.

Praktická část práce je rozdělená do dvou fází, a to vývoj prototypu adaptivního IDS založeného na strojovém učení a samotného penetračního nástroje, který je na vyvinutém IDS testován. Na obrázku 7 vidíme značně zjednodušený pohled na testovací prostředí, které se skládá ze dvou virtuálních strojů. První stroj s názvem Pentest VM zahrnuje webovou aplikaci, která obsahuje veškerou aplikační logiku penetračního nástroje. Webová aplikace disponuje REST API, které slouží pro interakci s nástrojem, a to jak ze strany uživatele, tak i ze strany druhého virtuálního stroje. Zároveň je na tomto stroji provozována i jednotka pro generování testovacího provozu, která je od webové aplikace oddělena. Druhý virtuální stroj nazvaný IDS VM zajišťuje chod vyvíjeného IDS a síťových služeb, na které je testovací provoz cílen. Provoz je zachycen IDS, které jej ohodnotí a pošle zpětnou vazbu penetračnímu nástroji, který na jejím základě optimalizuje parametry následně zaslaného provozu. Komunikace probíhá v reálném čase a uživatel může sledovat průběh učení ve webovém rozhraní.



Obrázek 7: Diagram komponent testovacího prostředí s vysokou úrovni abstrakce

7 Návrh a implementace IDS

Kapitola je v první části věnována analýze datové sady, na jejímž základě je vytvořena detekční jednotka IDS. Další část navazuje na analýzu a zabývá se principem zpracování zachyceného provozu a implementací samotného IDS.

7.1 Analýza datové sady

Analýza se zabývá datasetem UNSW-NB15 [76], který obsahuje data popisující síťový provoz. V datasetu se vyskytuje jak legitimní provoz, tak útoky na zařízení v síti. Dataset je určen pro klasifikaci, případně detekci anomálií, s cílem rozlišit útoky od legitimního provozu. V oblasti analýzy síťového provozu existuje celá řada podobných datasetů (např. dobře známý KDD98 nebo KDDCUP99), jejich problémem je neaktuálnost dat, proto jsem zvolil UNSW-NB15, který byl vytvořen v roce 2015 a je dle mého názoru stále relevantní.

Data vznikla na základě simulace provozu v sítí pomocí nástroje IXIA PerfectStorm v Cyber Range Lab of the Australian Centre for Cyber Security (ACCS). Ze 100 GB zachyceného provozu ve formě pcap souborů byl pomocí nástrojů Argus a Bro-IDS vytvořen tento dataset, který popisuje jednotlivé datové toky pomocí vektorů s 49 prvky. K dispozici je cca 2,5 miliónů záznamů v CSV, v rámci analýzy je použita pouze část těchto dat, která jsou k dispozici ke stažení jako trénovací, resp. testovací, data. Cílem analýzy je zjistit, zda je možné takto reprezentovaná data využít v rámci adaptivního IDS.

7.1.1 Explorační analýza

Trénovací sada obsahuje 175 341 záznamů, testovací 82 332. Rozlišujeme 10 kategorií provozu, běžný provoz a 9 typů útoků, výčet vidíme v Tabulce 2. Dataset obsahuje i binární vyjádření třídy, kdy rozlišujeme pouze mezi běžným provozem a útokem.

Tabulka 2: Třídy provozu

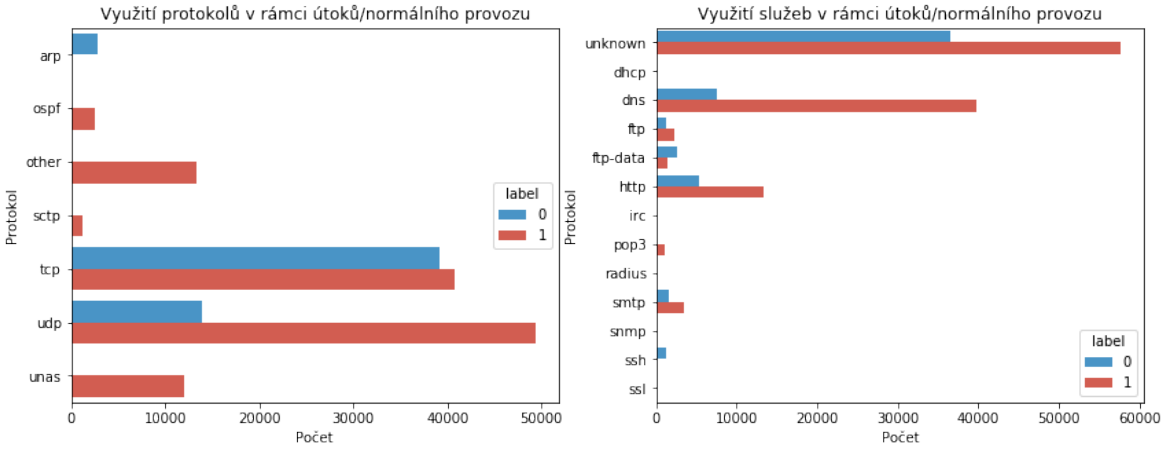
Kategorie provozu
Normal
Fuzzing
Analysis
Backdoor
DoS
Exploit
Generic
Reconnaissance
Shellcode
Worms

Datová sada obsahuje celkem 49 atributů, jejichž kompletní výčet a popis je uveden v Tabulce 8 a 9 v příloze. V datech se vyskytují numerické i kategoriální atributy. V trénovací a testovací sadě jsou odstraněny atributy týkající se IP, portu a času počátku/konce spojení, protože při použití těchto atributů by došlo k jevu, který se označuje jako tzv. information leak, který je zapříčiněn využíváním atributů v trénovací sadě, jejichž hodnotu v testovací sadě nemáme k dispozici. Tento jev způsobuje overfit modelu. Příkladem overfitu v tomto případě by byla např. identifikace útočníka na základě zdrojové IP, kdy v trénovací sadě by tento přístup určitě fungoval (jednalo se o malou testovací síť), ale v praxi by zcela jistě selhal, protože pro útoky se nejčastěji využívá botnet s různými IP. Pro ukázku vidíme výběr několika atributů v Tabulce 3.

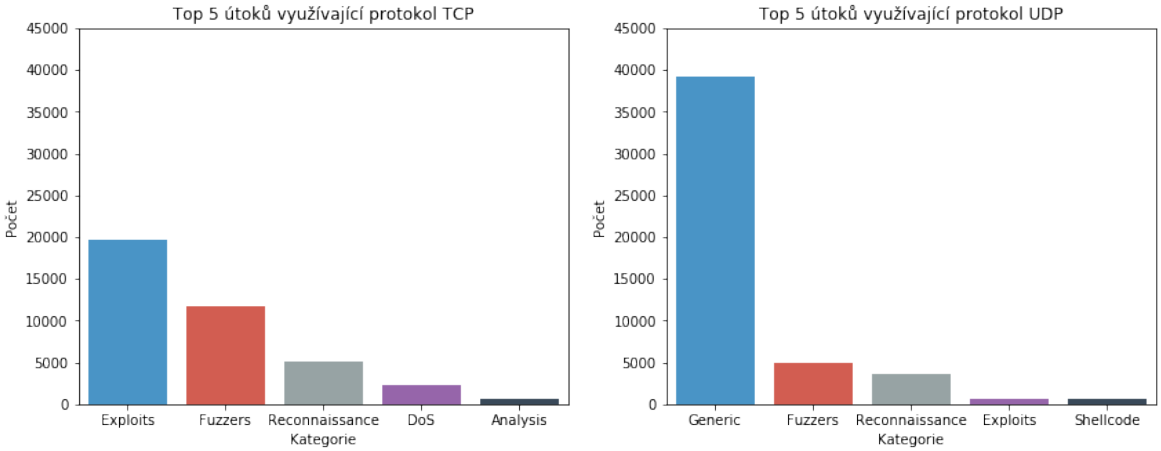
Tabulka 3: Ukázka atributů datové sady

Číslo	Název	Typ	Popis
1	srcip	Nominální	Zdrojová IP
2	sport	Celočíselný	Zdrojový port
3	dstip	Nominální	Cílová IP
4	dsport	Celočíselný	Cílový port
5	proto	Nominální	Protokol
6	dur	Spojitý	Trvání spojení
7	sbytes	Celočíselný	Počet odeslaných bytů
8	dbytes	Celočíselný	Počet přijatých bytů
9	sloss	Celočíselný	Počet ztracených odeslaných paketů zdrojem

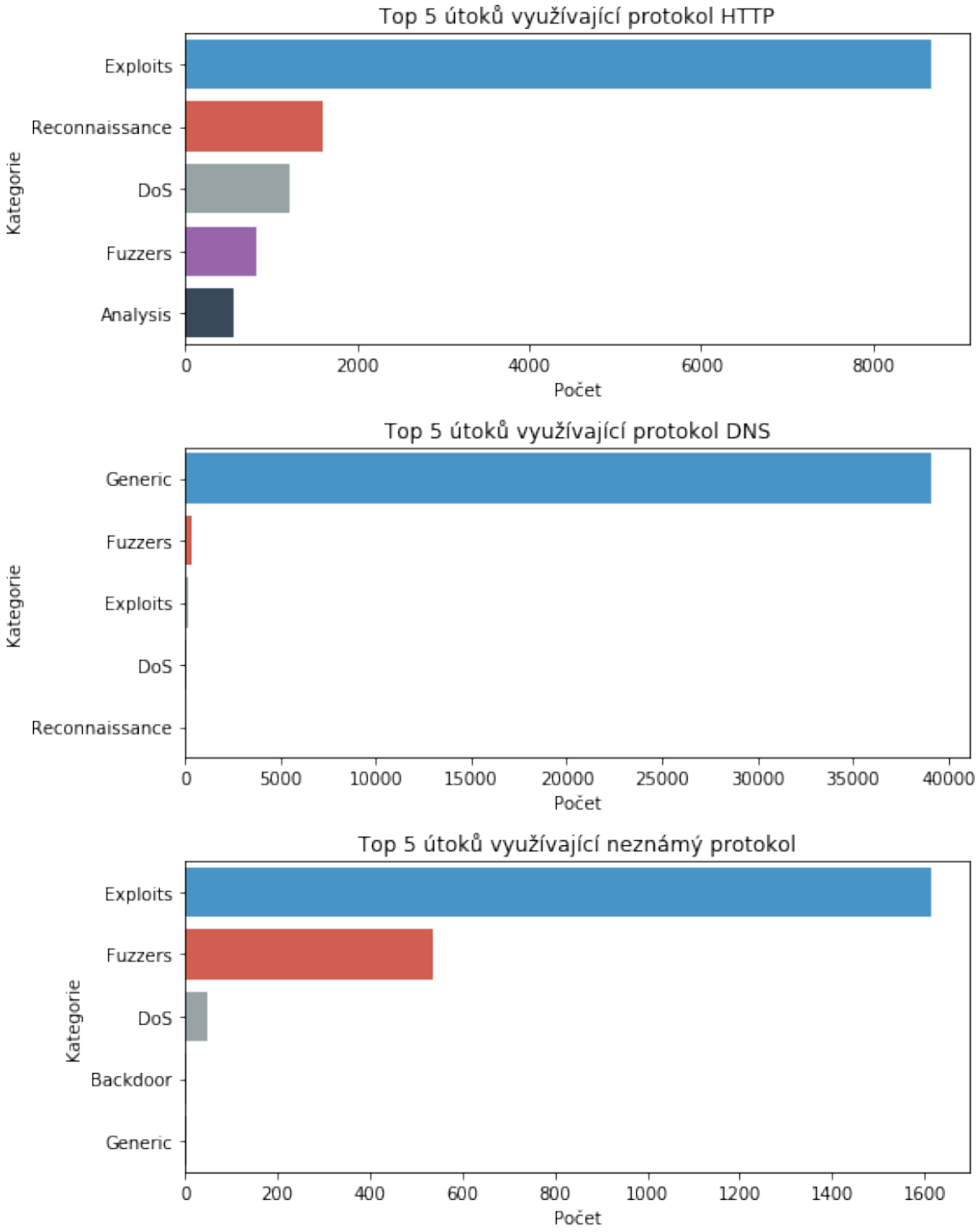
Na Obrázcích 8, 9 a 10 vidíme jaké protokoly a služby jsou nejvíce používané pro útok a jaký je podíl normálního provozu a útoků v rámci těchto protokolů. Modrou barvou je vyznačen legitimní provoz (label je 0), červená značí útok (label je 1). Velmi populární jsou útoky využívající protokol UDP, z toho plyne, že je i velký podíl útoků v rámci protokolů aplikační vrstvy, které UDP používají. Velmi populární jsou pro útoky protokoly DNS a HTTP, bohužel u velké části provozu nebyl protokol aplikační vrstvy detekován. Útokům využívající protokol TCP dominují exploity a fuzzing, což jsou stále poměrně běžné útoky, které plynou ze zastaralých verzí webových serverů a špatné validace vstupů. U UDP zaujímá největší podíl "generický"útok, bohužel v popisu datové sady není uvedeno, jaké útoky do této kategorie spadají. Ale jak si můžeme všimnout na Obrázku 10, celá tato skupina využívá protokol DNS, dle mého názoru se může jednat např. o DNS reflection DoS útok. Situaci ohledně častého výskytů exploitů v rámci útoku dokládají i nejčastější útoky využívající protokol HTTP, viz Obrázek 10.



Obrázek 8: Využití protokolů a služeb v rámci provozu



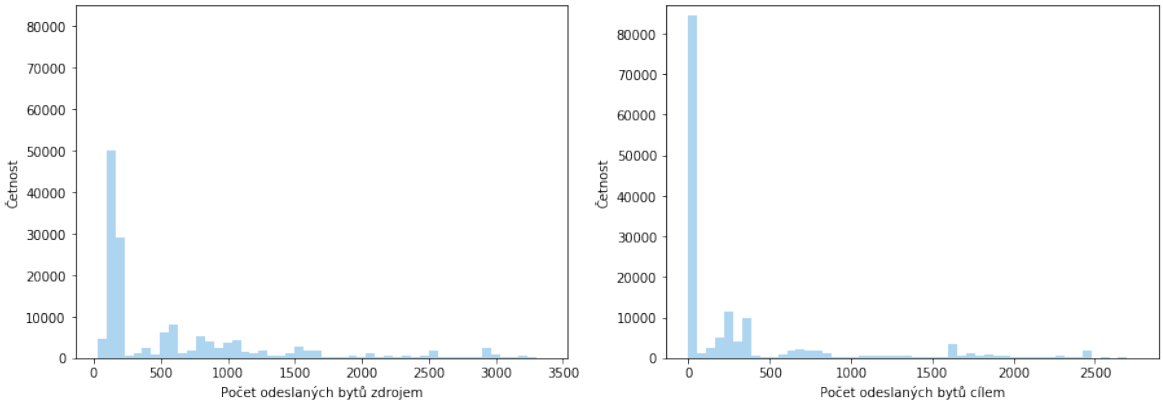
Obrázek 9: Nejčastější typy útoků pro protokoly TCP a UDP



Obrázek 10: Nejčastější typy útoků pro protokoly HTTP, DNS a nespecifikovaný protokol

V rámci numerických atributů jsou data značně pozitivně zešikmená a navíc obsahují velké množství odlehlých pozorování, což dokládá Obrázek 11. Explorační analýza numerických atributů je provedena pro data po odstranění odlehlých pozorování pomocí metody interkvartilového

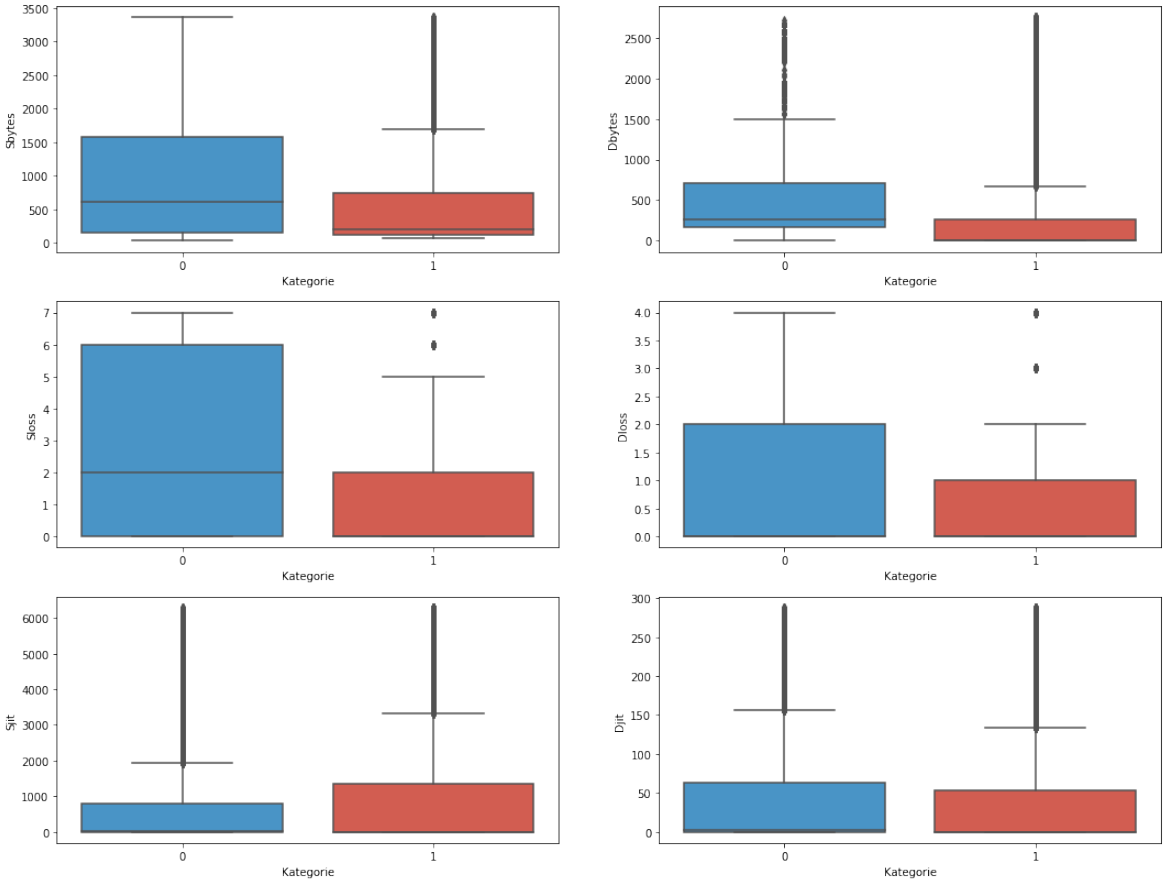
rozpětí (IQR). Výběrové charakteristiky jsou uvedeny v Tabulce 4. Z grafů a tabulky je patrné, že když srovnáme histogramy počtu odeslaných bytů v požadavcích a odpovědích, velmi často dochází k absenci odpovědi na požadavek. Na základě výběrových kvantilů lze usoudit, že žádosti jsou navíc náročnější na objem přenesených dat.



Obrázek 11: Histogramy počtu odeslaných bytů

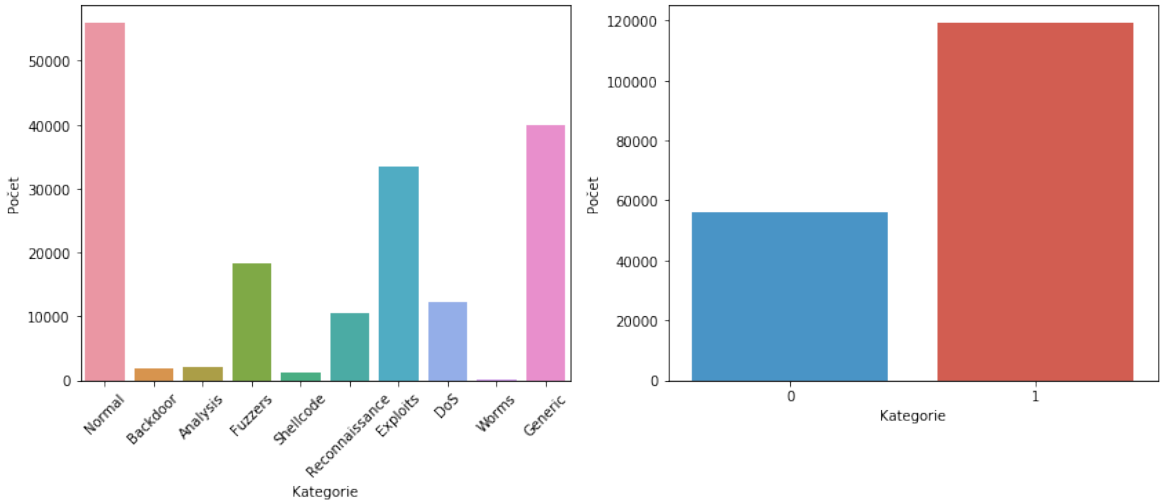
Tabulka 4: Výběrové charakteristiky vybraných atributů

Míry polohy	sbytes	dbytes	sloss	dloss	sjit	djit
Minimum	28	0	0,0	0,00	0	0
Dolní kvantil	114,00	0	0,0	0,00	0	0
Medián	200,00	0	0,0	0,00	0	0
Průměr	512,00	186	0,6	0,41	713	25
Horní kvantil	564,00	268	0,0	0,00	18	0
Maximum	336,00	2754	7,0	4,00	6283	287
Míry variability						
Směr. Odchylka	510	410	1,1	0,92	1500	55



Obrázek 12: Krabicové grafy vybraných atributů

Na Obrázku 12 vidíme krabicové grafy vybraných atributů, které jsou vztaženy ke třídě provozu, 0 opět znamená běžný provoz a 1 útok. Krabicové grafy počtu přenesených bytů odpovídají závěrům učiněným na základě výběrových charakteristik, běžný provoz vykazuje mnohem větší rozptyl. Ztrátovost bývá u útoků nižší než je tomu u běžného provozu, to stejné platí i o rozptylu hodnot. Jitter vykazuje velmi podobné charakteristiky u obou typů provozu, nicméně u odpovědí na požadavky je jeho hodnota obecně mnohem nižší.



Obrázek 13: Vyváženost tříd

Obrázek 13 zobrazuje rozdělení provozu dle typu, vidíme že v datové sadě je dvakrát více instancí reprezentujících útok, než instancí, které představují běžný provoz, třídy tedy nejsou vyvážené. Co se týká rozdělení typů útoků, velmi často se v datové sadě vyskytují exploity, fuzzing, scanování a DoS útoky, velká část instancí spadá do třídy generických útoků, které jsou zmíněny dříve v textu.

Data neobsahovala žádné chybějící hodnoty a pouze několik atributů je nominálních. Pro nominální atributy bylo provedeno sloučení málo četných hodnot do jedné třídy a následovalo převedení na binární vektory metodou One-Hot encoding, binarizace nominálních hodnotu umožňuje využít algoritmy, které umí pracovat jen s numerickými hodnotami.

7.1.2 Srovnání klasifikačních algoritmů

V rámci klasifikace byly použity algoritmy Naive Bayes (NB), KNN, Random forest (RF), Light gradient boosting (LGB) a Adaboost (ADA). Srovnání proběhlo využitím algoritmů na původních datech a po provedení resamplingu metodami Random resamplig a SMOTE, které umožňují vyvážit počet instancí v jednotlivých třídách. Klasifikace probíhá binárně, rozlišuje se pouze mezi legitimním provozem a útokem. Pro každou metodu je provedena 10-fold cross validace, zvolenou metrikou je přesnost, ta je dána vztahem (2). Následně je klasifikační model aplikován na testovací data, pro zjištění přesnosti na instancích, se kterými se model dosud nesetkal.

$$Přesnost = \frac{Počet\ správně\ klasifikovaných\ instancí}{Celkový\ počet\ instancí} \tag{2}$$

Tabulka 5: Přesnosti algoritmů pro 10-fold CV a testovací data

Algoritmus	Průměr (10-CV)	Směr. odchylka	Testovací data
Původní data			
NB	0,795	0,004	0,706
KNN	0,893	0,003	0,783
RF	0,957	0,001	**0,879**
LGB	0,956	0,002	0,875
ADA	0,940	0,003	0,849
Random resampling			
NB	0,769	0,004	0,718
KNN	0,862	0,003	0,785
RF	0,957	0,001	0,890
LGB	0,949	0,001	**0,911**
ADA	0,932	0,003	0,900
SMOTE resampling			
NB	0,764	0,004	0,723
KNN	0,866	0,002	0,783
RF	0,954	0,002	0,882
LGB	0,954	0,002	**0,889**
ADA	0,934	0,002	0,880

V Tabulce 5 vidíme výsledky srovnání zmíněných algoritmů. Ve všech případech poskytuje 10-fold crossvalidace optimističtější výsledky než ohodnocení na testovací sadě. Dle směrodatné odchylky lze usoudit, že přesnost algoritmů je v rámci crossvalidace velmi stabilní.

Pro původní data, kde nebyla aplikovaná žádná metoda pro vyvážení, fungoval nejlépe algoritmus Random Forest (RF). Po aplikaci resample metod se jako nejvhodnější jeví algoritmy založené na boosting ensemble přístupu. Nejlepších výsledků dosáhl algoritmus Light Gradient Boosting (LGB) na datech, které byly vyvážené pomocí random resample metody. Nevýhodou ensemble metod je vysoká časová náročnost vytvoření modelu, jak vidíme v tabulce 6, z tohoto hlediska je Naive-Bayes řádově rychlejší ve srovnání s ostatními metodami, což často představuje i přes nižší přesnost výhodu v oblasti real time systémů.

Tabulka 6: Srovnání času vytvoření modelu na původních datech

Algoritmus	Čas vytvoření modelu (s)
NB	**0,520**
KNN	22,120
RF	10,405
LGB	2,546
ADA	22,068

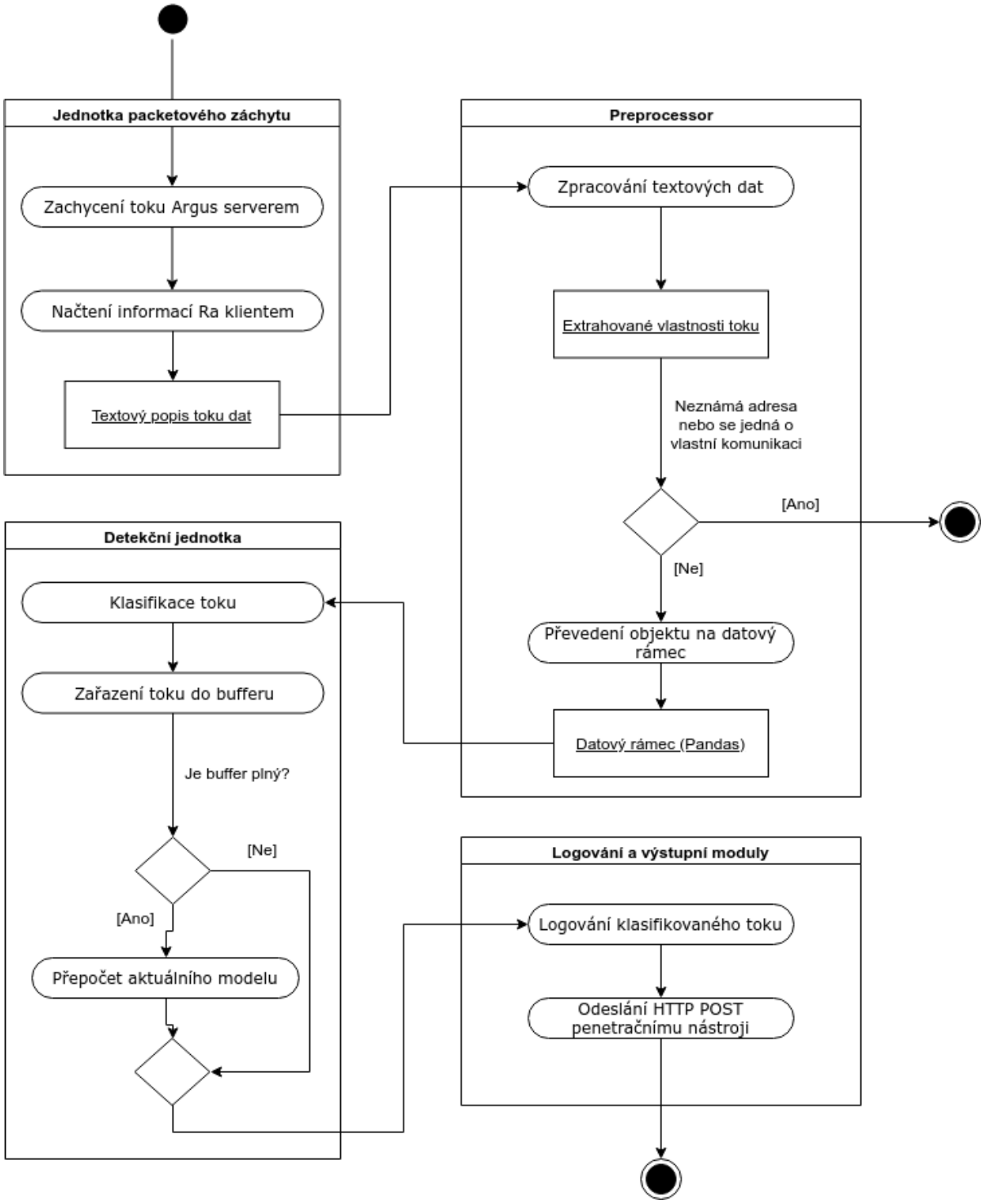
7.2 Vlastní implementace IDS

Analýza potvrdila, že reprezentace dat, která je použita v datové sadě UNSW-NB15, je vhodná pro klasifikační úlohy. Z toho důvodu je použita totožná reprezentace dat i v IDS.

IDS se zpravidla dají logicky rozdělit na čtyři části, kdy výstup předcházející části představuje vstup části následující. Stejně je tomu i u implementovaného IDS, rozlišujeme tedy tyto části:

- Jednotka paketového záchytu, která slouží pro zachycení síťového provozu.
- Preprocesor, který je zodpovědný za transformaci dat.
- Detekční jednotka, jejímž účelem je rozhodnout, do které třídy provoz spadá.
- Systém logování a výstupní moduly, které slouží pro zobrazení výstrah, případně integraci s dalšími nástroji.

Jednotka packetového záchytu je realizována prostřednictvím nástroje Argus [77]. Argus je open-source projekt, který je zaměřen na audit síťového provozu, používá architekturu klient-server. Server se nachází v komunikační trase mezi účastníky a zachytává síťový provoz, který agreguje do TCP, resp. UDP, toků dat. Poskytnuté informace jsou tedy souhrnem pro jednotlivá spojení, nikoli packety. Data jsou čtena prostřednictvím klientů, v práci je použit klient pojmenovaný `Ra`, který zprostředkovává přístup k informacím o zachycených tocích v reálném čase, nicméně v rámci projektu Argus je k dispozici celá řada dalších různě zaměřených klientů. Informace jsou vypisovány na standardní výstup. IDS tato data čte a textový výstup předává preprocessoru. Přijaté informace zahrnují informace o použitých portech, IP adresách účastníků, počtu přenesených bytů ve spojení, jitter a další statistiky. Argus byl použit také pro zachycení běžného provozu na virtuálním serveru IDS VM, který slouží jako počáteční datová sada, před započetím učení útoků vyvíjeným penetračním nástrojem.



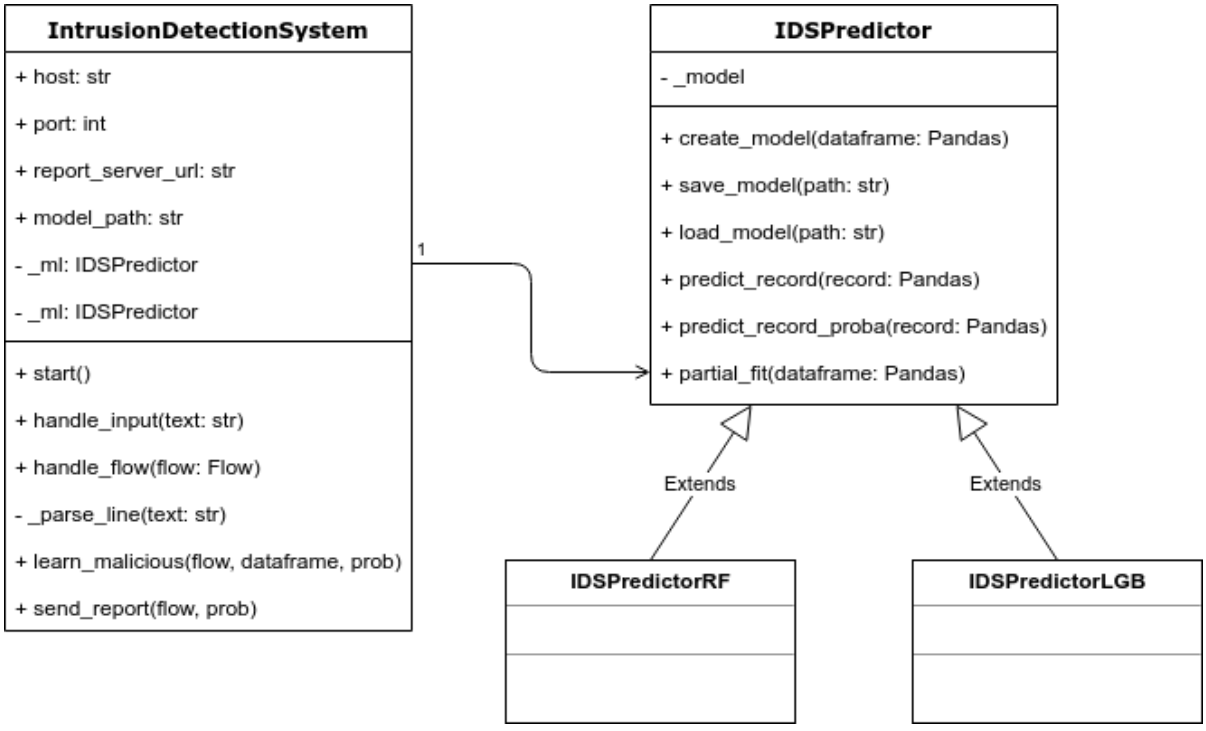
Obrázek 14: Diagram aktivit - zpracování síťového toku v IDS

Vstupem preprocessoru je textová reprezentace dat, která v prvním kroku projde parserem, který extrahuje potřebná pole a vytvoří z nich objekt. Následně dojde ke kontrole, zda extrahovaná data jsou validní, pokud ne zpracování toku končí. V případě úspěšné validace je objektová

reprezentace převedena na datový rámec, zde je využita knihovna Pandas [78], který je vstupem detekční jednotky.

Detekční jednotka využívá pro klasifikaci provozu strojové učení. Aktuálně je možné využít klasifikátory Naive Bayes, Random Forest [79] a Light Gradient Boosting, viz kapitola 7.1. Poslední dva zmíněné algoritmy patří mezi ensemble metody, z důvodu rychlosti přepočtu modelu se používají řádově pouze jednotky dílčích klasifikačních modelů. Model není přepočítáván po každém toku, ale toky jsou zařazovány do bufferu. Identifikace provozu, který slouží k učení probíhá na základě IP adresy. Veškerý provoz, jehož zdrojová IP adresa odpovídá nastavené adrese penetračního nástroje, je považován za útok, provoz z ostatních IP adres je brán jako legitimní. Po naplnění bufferu je model přepočítán a klasifikovaný provoz je předán modulu logování a výstupních modulů.

Vyvinutý IDS pracuje v tzv. online režimu, informace o analyzovaném provozu jsou tedy vypisovaný na standardní výstup. Modul logování je zodpovědný rovněž za komunikaci s penetračním nástrojem. Komunikace probíhá prostřednictvím REST API. Jediný požadavek na IDS, který má s nástrojem spolupracovat, je schopnost odeslat HTTP POST, předávající nástroji informaci o tom, s jakou pravděpodobnosti je zaslaný provoz vyhodnocen jako škodlivý. Celý popsaný proces je následně opakován pro všechny analyzované toky dat. Vizualizace procesu se nachází na Obrázku 14. Na Obrázku 15 vidíme třídní diagram IDS.



Obrázek 15: Třídní diagram - komponenty IDS

8 Návrh a implementace penetračního nástroje

Jak je zmíněno v kapitole 7.2, detekční jednotka testovacího IDS je založena na strojovém učení a poskytuje zpětnou vazbu v podobě pravděpodobnosti, se kterou je analyzovaný provoz klasifikován jako škodlivý.

Cílem penetračního nástroje je nalézt takový (nelegitimní) provoz, který je v daném čase klasifikován jako útok s nízkou pravděpodobností. Nástroj tak umožňuje odhalení nedokonalostí klasifikačního modelu, které mohou být následně eliminovány tím, že dojde k přepočtu a vytvoření nového modelu, který již bude provoz schopen klasifikovat správně. Díky tomu, že IDS je schopen poskytovat penetračnímu nástroji zpětnou vazbu, je možné na úlohu pohlížet jako na optimalizační problém s dynamickou účelovou funkcí.

Z tohoto důvodu penetrační nástroj využívá při své činnosti vybrané biologicky inspirované algoritmy, které jsou vhodné k řešení nekonvexních optimalizačních problémů. Dalším důvodem pro volbu této třídy algoritmů je schopnost řešit optimalizační problémy, které nemají analyticky definovanou účelovou funkci, což je výhodou i v tomto případě, protože IDS neposkytuje žádné informace o vypočteném modelu, ale známe pouze jeho reakci na daný vstup. Algoritmus 2 poskytuje zjednodušený pohled na činnost penetračního nástroje. Přesný postup se může lehce lišit v závislosti na zvoleném algoritmu, nicméně princip zůstává vždy stejný.

Vstup: IP adreasa IDS, TCP a UDP porty služeb, Hyperparametry algoritmu

1 **while** aktuální_iterace $<$ počet_iterací **do**
2 **foreach** *jedinec v populaci* **do**
- Aplikace operátorů algoritmu pro vytvoření nového potomka;
- Zajištění, že parametry potomka splňují definované meze;
- Transformace parametrů potomka na parametry síťového provozu;
- Předání parametrů generátoru provozu;
- Ohodnocení potomka na základě zpětné vazby;
- Aktualizace dosavadní populace;
- Odeslání informací o průběhu;

3 **end**
4 **end**

Algoritmus 2: Průběh testování s vysokou úrovni abstrakce

8.1 Komponenty penetračního nástroje

Penetrační nástroj je tvořen třemi komponentami. Následující podkapitoly se zabývají popisem stěžejních částí jednotlivých komponent a jejich propojením.

První komponentou je webová aplikace, která obsahuje veškerou aplikační logiku a poskytuje data ostatním komponentám. Webová aplikace je vytvořená v programovacím jazyce Java a je postavená na Spring Boot frameworku s architekturou MVC. Komunikace probíhá primárně prostřednictvím REST API. Druhou komunikační trasu představují Websockety, které jsou použity v případě, že je nutné iniciovat přenos dat ze strany serveru. Daný framework byl zvolen zejména díky snadné integraci REST API a Websocketů v rámci jedné aplikace a také možnosti snadné rozšiřitelnosti o novou funkcionalitu.

Druhou komponentu představuje generátor provozu, který je posílán na testovaný IDS. Generátor je vytvořen v programovacím jazyce Python. Data jsou posílány z webové aplikace pomocí technologie Websocket formou JSON modelů, ze kterých je následně vytvořen odesílaný provoz.

Třetí část tvoří uživatelské rozhraní. To je vytvořené jako single-page aplikace v Javascriptu. Rozhraní slouží pro konfiguraci parametrů testovacího procesu, zároveň poskytuje i aktuální informace o jeho průběhu. Rozhraní je napojeno na zmíněné REST API, o stavu testování je uživatel informován v reálném čase s využitím technologie Websocket.

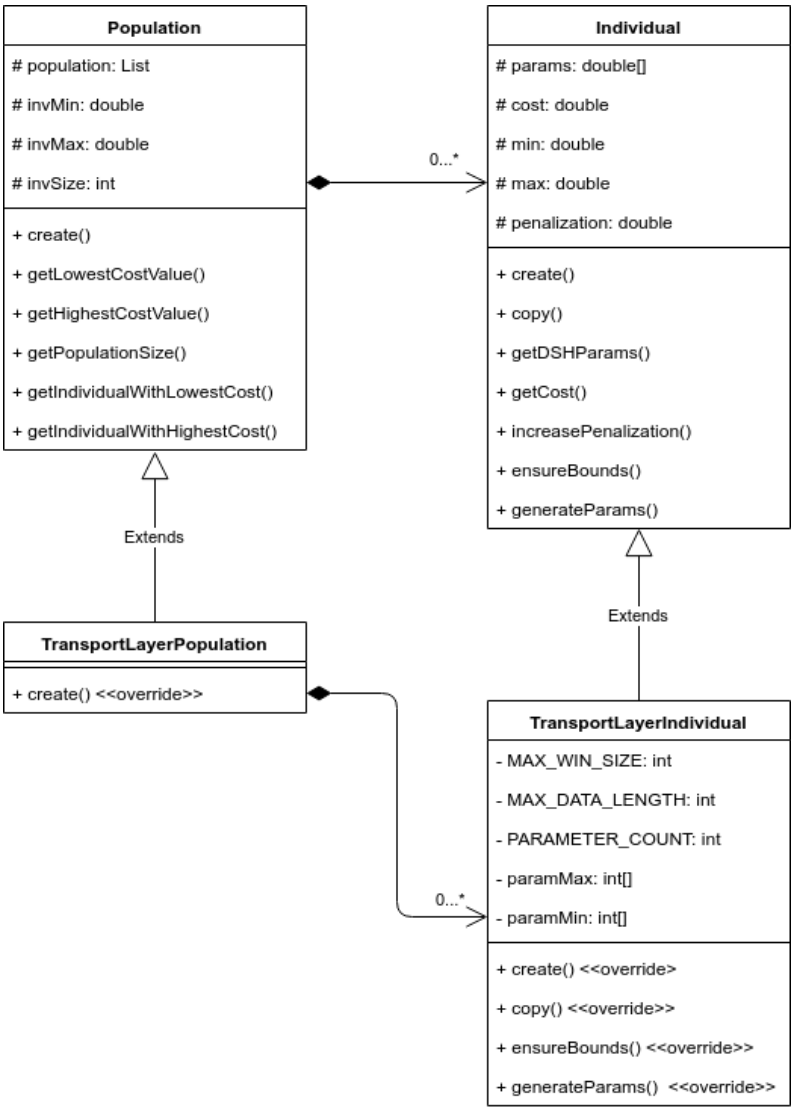
8.2 Reprezentace jedinců

Nástroj implementuje dva přístupy k optimalizaci provozu, oba přistupují k reprezentaci jedinců jinak. Společným rysem je, že evoluce pracuje s parametry datagramů na úrovni transportní vrstvy modelu TCP/IP. Vytvořené datagramy se liší velikosti sliding window, nastavenými TCP příznaky a množstvím odeslaných dat. Zmíněné parametry byly zvoleny na základě možností popsaných nástrojů v rámci kapitoly 4, které jsou rovněž určeny ke generování síťového provozu.

První přístup je založen na principu discrete set handling (DSH). Jedinec je tvořen polem reálných čísel, evoluce tedy probíhá v tomto prostoru. Čísla jsou následně zaokrouhlena a slouží jako ukazatel na prvky v množině možných datagramů. Počet parametrů jedince volí uživatel a jejich počet odpovídá počtu odeslaných datagramů v rámci ohodnocení jednoho jedince. V aplikaci tuto reprezentaci zajišťují třídy `Population` a `Individual`. Třída `Population` obsahuje seznam jedinců a poskytuje rozhraní pro výběr jedinců dle fitness, vytvoření nové populace dle zadaných parametrů apod. Třída `Individual` reprezentuje jednotlivé jedince. Kromě udržování hodnot parametrů a ohodnocení slouží třída také k zajištění toho, že hodnoty parametrů patří do vymezeného intervalu. Funkcionalitu zajišťuje metoda `ensureBounds`, která nahradí hodnoty mimo interval náhodnými hodnotami z intervalu. Spodní hranice je nulová, horní je dána počtem prvků v uspořádané množině datagramů. Výhodou tohoto přístupu je snadná rozšiřitelnost o nové vzory provozu, které se nemusí v budoucnu omezovat pouze na parametry transportní vrstvy, ale může se jednat např. o útoky využívající protokoly aplikační vrstvy.

Druhá reprezentace pracuje přímo s parametry datagramů. Jedinec je opět tvořen polem reálných čísel, která jsou před mapováním na parametry datagramu zaokrouhlena. Parametry jsou vždy tři, a to TCP příznak, velikost dat a velikost sliding window. Jeden jedinec odpovídá jednomu datagramu. Reprezentace je realizována třídami `TransportLayerPopulation` a `TransportLayerIndividual`. Tyto třídy jsou potomky výše zmíněných tříd. Možné operace jsou shodné, ale liší se jejich implementace. Příkladem je metoda `ensureBounds`, která nyní musí zajišťovat, že např. velikost sliding window nepřesáhne mez definovanou v RFC.

Na Obrázku 16 vidíme třídní diagram popsané části aplikace vizualizující popsané třídy a vazby mezi nimi.



Obrázek 16: Třídní diagram - reprezentace jedinců

8.3 Účelová funkce

Pokud je použit přístup discrete set handling, každý parametr jedince reprezentuje jeden datagram. IDS ohodnocuje pravděpodobností útoku každý datový tok zvlášť a cílem účelové funkce je agregace těchto pravděpodobností k ohodnocení jedince. Účelová funkce je definována vzorcem 3, vstupem je n-tice pravděpodobností získaná z odpovědí IDS. Hodnota α_i je penalizační faktor jedince určen na základě jeho stáří. Zavedení penalizace je důsledek závislosti získaných pravděpodobností na stavu modelu. Je potřeba si uvědomit, že pokud bude ohodnocen datagram v čase t_0 na základě modelu, který dosud neobsahuje vzor pro jeho správnou detekci, bude obdržená pravděpodobnost nulová. Pokud stejný datagram ohodnotí IDS v čase t_1, ve kterém již model potřebný vzor obsahuje, bude datagram ohodnocen vyšší hodnotou pravděpodobnosti. Absence penalizace by způsobila, že jedinci, kteří byli ohodnoceni na počátku testovacího procesu, budou mít přiřazenou mnohem lepší hodnotu fitness, protože model je nebyl schopen správně klasifikovat. Tito jedinci by byli zvýhodněni, což by zamezilo jejich nahrazení potomky, což by naprosto zabránilo evoluci. Z toho důvodu je penalizační faktor zvyšován o konstantu (aktuální hodnota je 0,2) po každé iteraci, čímž je zajištěna diverzibilita populace. Algoritmy, které nejsou založeny na generacích jedinců a svým principem stagnaci zamezují, mohou penalizační faktor ponechat nulový po celou dobu činnosti. Tak je tomu např. u algoritmu Grey Wolf Optimization (viz kapitola 5.1.1.3).

$$f_{dsh}(p_1, p_2, ..., p_n) = \frac{\sum_{i=1}^{n} p_i}{n} + \alpha_i \tag{3}$$

V případě přímé optimalizace parametrů datagramů se účelová funkce značně zjednoduší, protože se pracuje jen s přiřazenou pravděpodobnosti pro daný datagram. V takovém případě je zjednodušená účelová funkce popsána vzorcem 4. Princip penalizace zůstává stejný.

$$f_{transport}(p) = p + \alpha_i \tag{4}$$

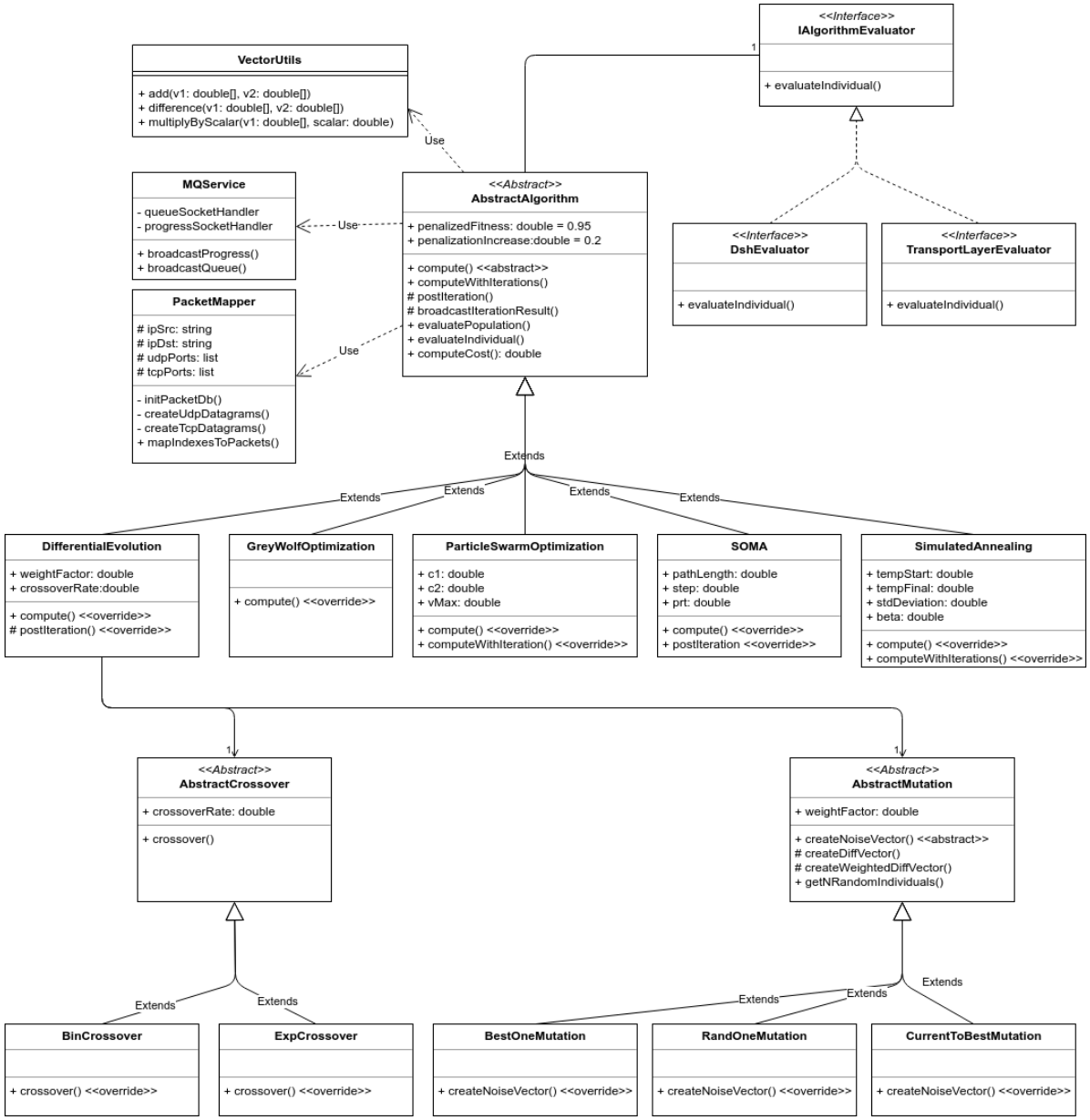
8.4 Moduly optimalizačních algoritmů

Třídní diagram modulů zodpovědných za řízení optimalizačního procesu vidíme na Obrázku 17. Hlavním prvek je třída `AbstractAlgorithm`. Tato třída je bázovou třídou pro všechny implementované algoritmy a jejím hlavním účelem je zapouzdření přístupu k dalším komponentám aplikace, které jsou využívány v rámci potomků. Komponenty slouží např. k propagování aktualizací do rozhraní, komunikaci s generátorem provozu nebo mapování hodnot parametrů na parametrizované datagramy. Za tyto činnost jsou zodpovědné třídy `MQService` a `PacketMapper`, které jsou detailně popsány v kapitolách 8.5 a 8.7. Dalšími důležitými třídami jsou `DshEvaluator` a `TransportLayerEvaluator`, které jsou zodpovědné za sběr dat z IDS a ohodnocení jedinců (proces je popsán v kapitole 8.6). S touto dvojicí tříd se pracuje v metodě `evaluateIndividual`, a to pouze na úrovni rozhraní `IAlgorithmEvaluator`, což zajišťuje, že implementace konkrétních algoritmů není závislá na reprezentaci jedinců. Metoda `compute` slouží k výpočtu jedné iterace

algoritmu, v metodě `computeWithIterations` je volána cyklicky pro zvolený počet iterací. Metoda `compute` je abstraktní a představuje naprosté minimum, které musí konkrétní algoritmy implementovat. Metoda `broadcastIterationResult` slouží k propagaci informací o průběhu do uživatelského rozhraní a pro tuto činnost využívá služeb třídy `MQService`. Poslední dvě důležité metody jsou `postIteration` a `computeCost`. První zmíněná je tzv. life-cycle metoda, která umožňuje podtřídě definovat operaci, která bude provedena po každé iteraci. Algoritmy založené na generacích ji využívají pro inkrementaci penalizačního faktoru jedince. Druhá zmíněná metoda implementuje výpočet účelové funkce definované v kapitole 8.3 a je zpřístupněna formou reference ve třídách implementující rozhraní `IAlgorithmEvaluator`. Implementované algoritmy často pracují s vektory, základní operace s nimi poskytuje třída `VectorUtils`. Jedná se zejména o součet vektorů a násobení skalárem, které jsou základními operacemi ve vektorovém prostoru a slouží jako základní stavební kameny pro snadnou implementaci složitějších operací ve vybraných algoritmech. Všechny zmíněné třídy, které jsou třídou `AbstractAlgorithm` využívány, nejsou instanciovány přímo třídou `AbstractAlgorithm`, ale je dodržen princip Inversion of Control a instance tříd jsou zpřístupněny pomocí tzv. constructor-based dependency injection. Přístup je zvolen s ohledem na možnost dalšího rozšíření aplikace.

V nástroji je implementovaných celkově pět biologicky inspirovaných algoritmů, princip fungování je popsán v kapitole 5.1.1. Každý algoritmus je reprezentován svou třídou, která dědí z bázové třídy `AbstractAlgorithm`. Hyper-parametry jednotlivých algoritmů jsou uloženy v instančních proměnných těchto tříd. Prvním algoritmem je Diferenciální evoluce, který implementuje třída `DifferentialEvolution`. V třídním diagramu vidíme, že třída asociuje dvojici tříd `AbstractCrossover` a `AbstractMutation`. Tyto třídy zajišťují společnou funkcionalitu pro různé strategie křížení a mutace. V nástroji je implementováno binomické a exponenciální křížení. Strategie mutací jsou implementovány tři, a to Best-One, Rand-One a Current-To-Best. Navržená hierarchie tříd umožňuje snadné rozšíření o další způsoby křížení a mutace a odstraňuje závislost třídy `DifferentialEvolution` na konkrétních implementacích zmíněných operátorů. Dále jsou implementovány hejnové algoritmy GWO, PSO a SOMA. Implementace algoritmu SOMA je postavena na strategii All-To-One. Posledním algoritmem je Simulované žíhání, implementovaný plán redukce teploty je popsán vzorcem 5.

$$t_{i+1} = \frac{t_i}{1 + \beta \cdot t_i} \tag{5}$$



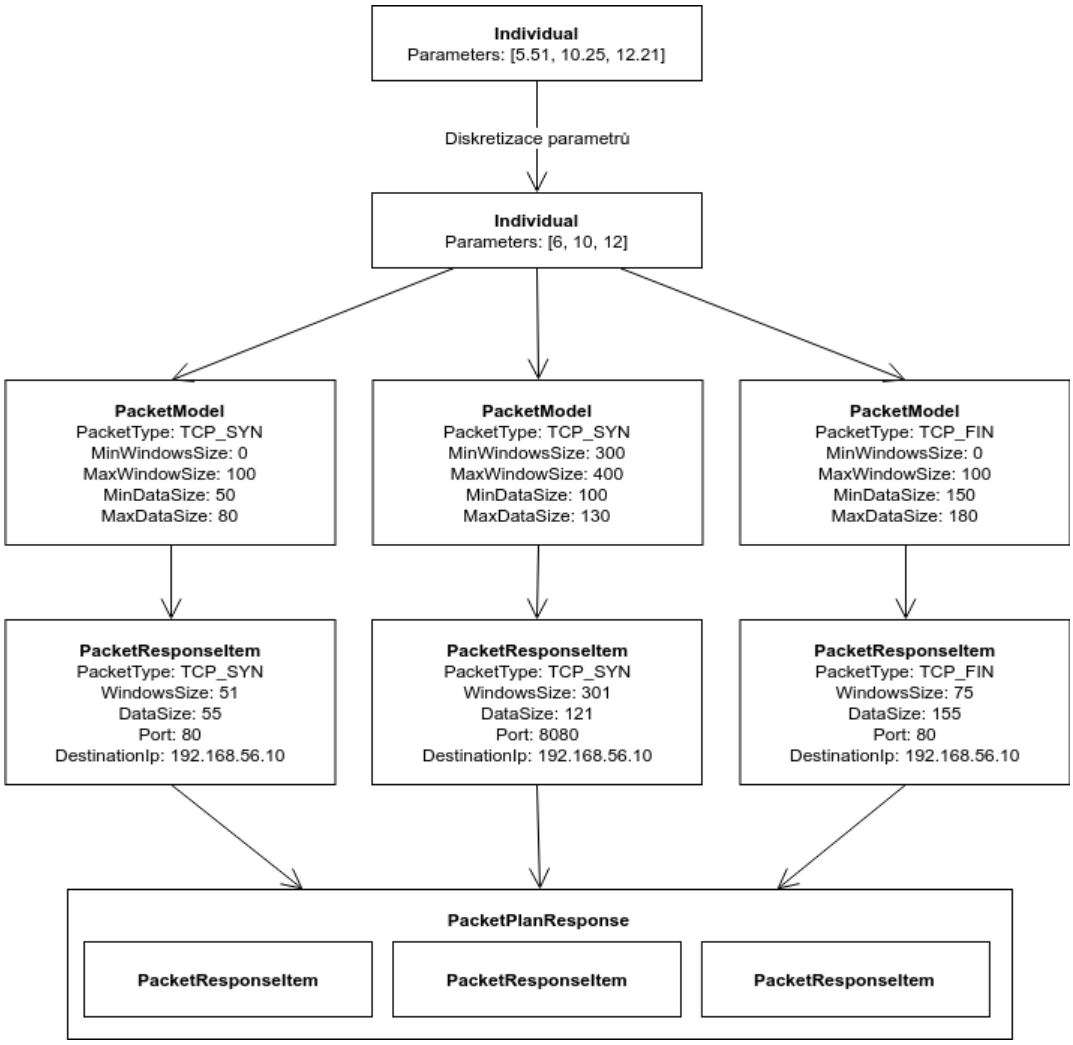
Obrázek 17: Třídní diagram - moduly optimalizačních algoritmů

8.5 Transformace jedinců na modely síťového provozu

Převod parametrů jedince na model provozu závisí na použité reprezentaci. Za celý proces je v nástroji zodpovědná třída `PacketMapper`, kterou vidíme na Obrázku 17. Metoda `initPacketDb` slouží k vytvoření definovaných vzorů datagramů, výsledné mapování parametrů na vzory implementuje metoda `mapIndexesToPackets`.

Na Obrázku 18 vidíme průběh mapování pro reprezentaci používající princip discrete set handling, v ukázce předpokládáme, že hodnoty parametrů nepřekračují definované meze. Prvním krokem je diskretizace hodnot parametrů. Celočíselné parametry jsou následně použity

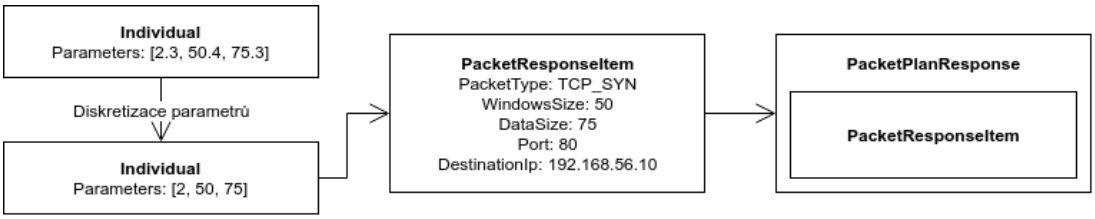
jako ukazatele do množiny vzorů, výsledkem je objekt typu `PacketModel`. Vzory nespecifikují přesné hodnoty velikosti odeslaných dat nebo sliding window, ale definují intervaly, kterých mohou hodnoty nabývat. Tímto způsobem je značně omezen počet potřebných vzorů v množině, čímž se zmenšuje také prohledávaný prostor, který je omezen právě počtem vzorů. Dalším krokem je vygenerování náhodných hodnot pro velikost odesílaných dat a sliding window. Hodnoty se generují z intervalů, které jsou definovány zmíněným vzorem, výsledkem je objekt typu `PacketResponseItem`. Posledním krokem je vytvoření objektu typu `PacketPlanResponse`, který slouží k seskupení objektů z předchozího kroku. Tento objekt je následně převeden na JSON a odeslán generátoru provozu, který jej zpracuje.



Obrázek 18: Transformace parametrů jedince na model provozu - metoda DSH

V případě, že se používá přístup, který optimalizuje parametry datagramu, je celý proces mnohem jednodušší. Diagram vidíme na Obrázku 19. První dva kroky jsou shodné s předchozím přístupem, rozdíl nastává ve třetím kroku. Neprovádí se transformace na `PacketModel`,

ale první hodnota určuje typ datagramu, druhá a třetí hodnota určuje velikost sliding window (v případě, že se jedná o TCP provoz) a velikost odeslaných dat. Výsledkem je opět `PacketPlanResponse`, který nyní sice obsahuje pouze jeden objekt, ale je zajištěna konzistence rozhraní pro komunikaci s generátorem provozu. Za tento přístup k mapování je zodpovědná třída `TransportLayerPacketMapper`, která je potomkem třídy `PacketMapper`. Takto navržené uspořádání tříd umožňuje v budoucnu snadné rozšíření o novou funkcionalitu, pokud dojde ke změně významu parametrů jedince, stačí pouze vytvořit nového potomka třídy `PacketMapper` a překrýt metodu `mapIndexesToPackets`, stejně jako je tomu u nynějších dvou implementací.

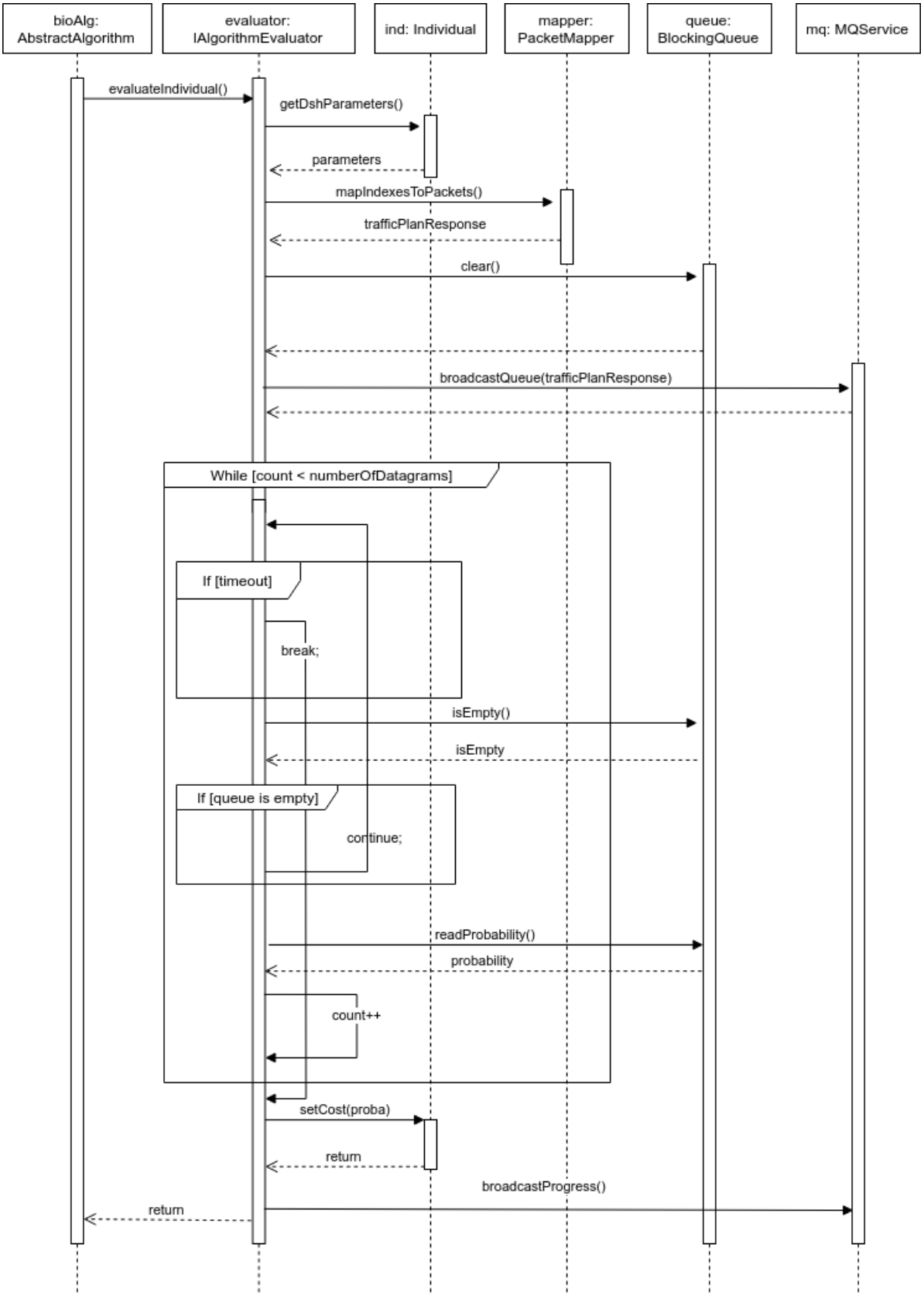


Obrázek 19: Transformace parametrů jedince na model provozu - optimalizace parametrů datagramu

8.6 Ohodnocení jedinců

Sekvenční diagram na Obrázku 20 popisuje průběh ohodnocení jedince. Průběh je popsán pro reprezentaci jedinců, která využívá množinu vzorů (DSH přístup). Prvním krokem je diskretizace parametrů a jejich převod na modely datagramů, tento krok odpovídá popisu v kapitole 8.5. Následně jsou modely odeslány generátoru provozu. Za tuto činnost je zodpovědná třída `MQService`. V metodě `broadcastQueue` dojde k převodu objektové reprezentace na JSON a odeslání dat pomocí Websocketu. Následující cyklus `while` opakovaně čte data z fronty odpovědí, dokud nejsou přijaty a zpracovány ohodnocení všech odeslaných datagramů ze strany IDS. Fronta odpovědí je tvořena objektem třídy `BlockingQueue`, která zajišťuje, že nedojde k problémům souběhu ve chvíli, kdy s kolekcí pracuje více vláken. Problémy souběhu představují reálné riziko, protože proces ohodnocení probíhá asynchronně. Celý optimalizační proces běží v samostatném vláknu, nicméně komunikace ze strany IDS využívá REST API, které rovněž používá vlastní vlákno. Bez použití tzv. thread-safe datové kolekce by mohlo dojít k situaci, že s ní pracuje jak vlákno REST API, tak vlákno optimalizačního procesu. V případě, že by došlo např. k chybě spojení a komunikace ze strany IDS byla neúspěšná, dochází po určitě době od poslední přijaté zprávy k tzv. timeoutu, který cyklus přeruší. Tento mechanismus brání uváznutí v cyklu. Po ohodnocení všech datagramů se použijí přijaté hodnoty pravděpodobností pro výpočet účelové funkce a finálnímu ohodnocení jedince. Celý proces se následně opakuje dokud nejsou ohodnocení všichni jedinci v populaci.

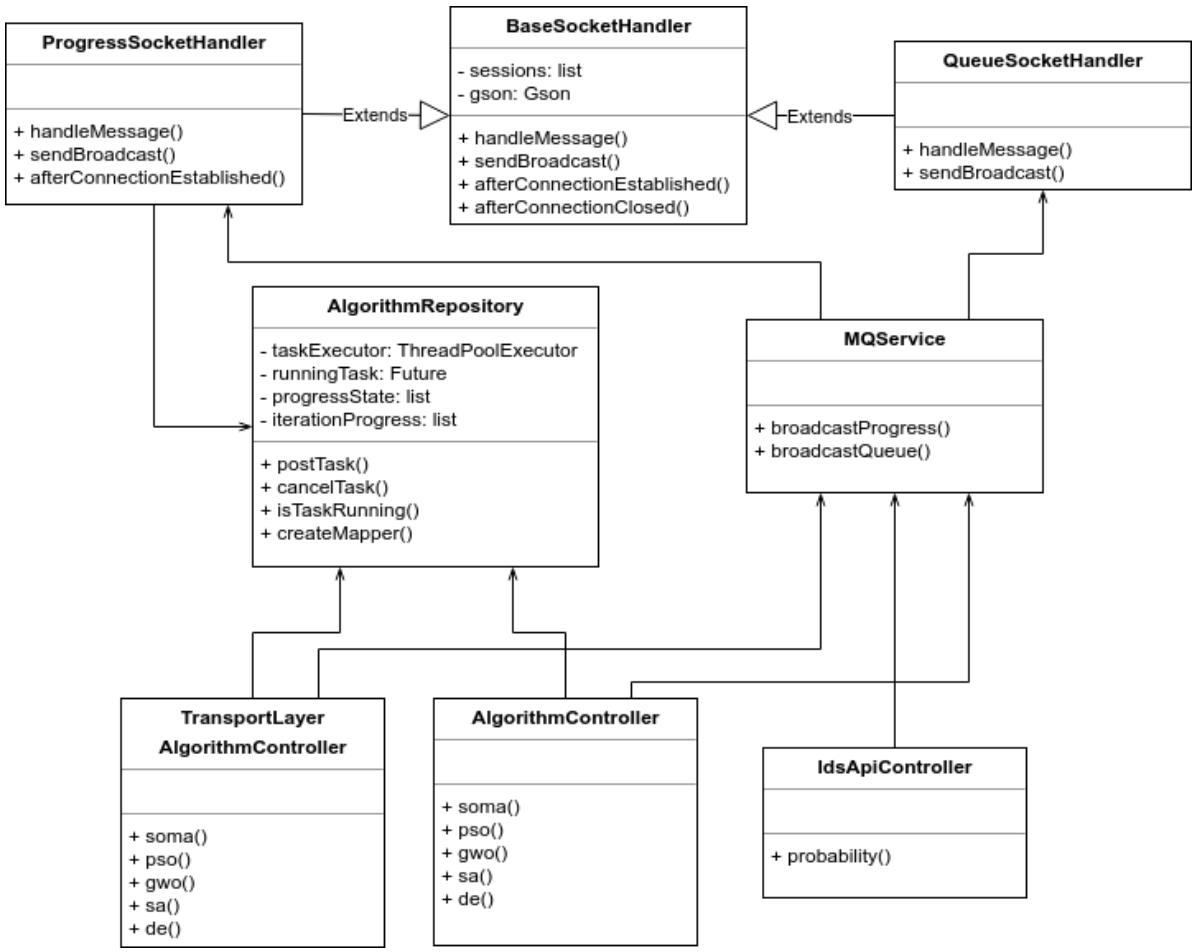
Pro reprezentaci, která je založená na přímé optimalizaci parametrů datagramu funguje ohodnocení analogicky. Jediným rozdílem je, že se ve `while` cyklu nečte sekvence pravděpodobností, ale pouze jediná hodnota.



Obrázek 20: Sekvenční diagram - ohodnocení jedince

8.7 Komunikace s dalšími komponentami nástroje

Třídní diagram na Obrázku 21 vizualizuje vztahy mezi třídami, které zajišťují komunikaci s generátorem síťového provozu a grafickým uživatelským rozhraním. V horní části digramu vidíme dvojici tříd `ProgressSocketHandler` a `QueueSocketHandler`, které jsou potomky třídy `BaseSocketHandler`. Tato trojice tříd zajišťuje real-time komunikaci s komponentami prostřednictvím komunikačního protokolu Websocket. První zmíněná třída zajišťuje přenos informací o průběhu testování, informace zahrnují např. parametry odeslaného provozu a jeho ohodnocení IDS. Data jsou následně zobrazována uživateli v rozhraní nástroje. K distribuci informací slouží metoda `sendBroadcast`, která odešle zprávu všem připojeným klientům k danému socketu. V případě, že se klient připojí k socketu v průběhu testovacího procesu, uplatní se metoda `afterConnectionEstablished` a jsou klientovi odeslány informace od začátku průběhu dodatečně. Třída `QueueSocketHandler` zabezpečuje odeslání modelů datagramů generátoru provozu. Bázová třída `BaseSocketHandler` zajišťuje základní funkcionalitu v podobě uchovávání informací o připojených klientech a zajišťuje také převod dat do formátu JSON, který je výhradním formátem odeslaných, resp. přijatých, zpráv.



Obrázek 21: Třídní diagram - komponenty zajišťující komunikaci s dalšími částmi nástroje

Ostatní části webové aplikace nepřistupují k potomkům třídy `BaseSocketHandler` přímo, ale komunikaci zastřešuje třída `MQService`. Třída `AlgorithmRepository` uchovává veškeré informace o průběhu testování. Tyto informace jsou odeslány v již zmíněné metodě `afterConnectionEstablished` nově připojeným klientům. Třída zároveň definuje rozhraní, které slouží k spuštění nového testování. K tomuto účelu slouží metoda `postTask`, která zajistí spuštění výpočtu v samostatném vlákně na pozadí, pool vláken je zpravován pomocí objektu třídy `ThreadPoolExecutor`. Třída rovněž umožňuje zjistit, zda aktuálně probíhá výpočet, případně jeho ukončení.

Ve spodní části diagramu vidíme MVC controllery, které zajišťují komunikaci přes REST API. Názvy metod korespondují s názvem cesty v URL. Třída `IdsApiController` definuje pouze jediný koncový bod, pomocí kterého předává IDS zpětnou vazbu nástroji, přijatá data z těchto HTTP POST požadavků jsou předána ke zpracování do fronty zpráv (viz kapitola 8.6). Zbylé dvě třídy `AlgorithmController` a `TransportLayerAlgorithmController` slouží pro spuštění testování z uživatelského rozhraní s příslušnými parametry, rozhraní opět přijímá HTTP POST požadavky.

```
@Data
public class SomaParameters {
    @Min(0)
    private double pathLength;
    @Min(0)
    private double step;
    @Range(min = 0, max = 1)
    private double prt;
}
```

Výpis 8: Třída s parametry algoritmu SOMA s anotacemi

Všechny uživatelské vstupy jsou validovány. Validace využívá JSR 380 anotace. Anotace jsou vztaženy k instančním proměnným a zahrnují např. definice rozsahu hodnot numerických proměnných. Kromě využití výchozích omezení je navíc v nástroji implementována i validace IP adresy prostřednictvím vlastní anotace. Ukázku vidíme ve Výpisu 8, třída v ukázce reprezentuje parametry algoritmu SOMA.

Výpis 9 obsahuje ukázku definice koncového bodu REST API pro spuštění algoritmu SOMA. V parametrech metody vidíme dvojici anotací `@Valid` a `@RequestBody`. První z nich slouží k validovaci dat požadavku, které se nachází v těle HTTP požavadku, což značí anotace `@RequestBody`. Výsledek validace je uložen do objektu `bindingResult`, který je druhým parametrem metody. V případě, že data nejsou validní, zpracování končí a klient obdrží HTTP odpověď `400 Bad Request` s hláškou, která pole nejsou validní. Pokud jsou data validní, zpracování pokračuje dále. Nejprve je vytvořená počáteční populace a instance příslušného algoritmu s parametry, které zadal uživatel. Následně je spuštěn testovací proces v samostatném vlákně (metoda `postTask`

zavolaná na objekt třídy `AlgorithmRepository`), jak je popsáno výše. Objekty tříd jako je např. `MQService` nejsou součástí volání metody, ale jsou dostupné prostřednictvím instančních proměnných třídy díky dependency injection, viz kapitola 8.4.

```
@PostMapping("soma")
public ResponseEntity soma(
@Valid @RequestBody
AlgorithmParameters<SomaParameters> parameters,
BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        ApiErrorsView body = errorConverter.convert(bindingResult);
        return new ResponseEntity<>(body, HttpStatus.BAD_REQUEST);
    }

    CommonParameters commonParameters = parameters.getCommonParameters();
    SomaParameters somaParameters = parameters.getAlgorithmParameters();
    PacketMapper mapper = algorithmRepository.createMapper(commonParameters);

    Population pop = new Population()
        .create(commonParameters.getNumberOfIndividuals(), 0, mapper.
            getMaximumIndex(), commonParameters.getNumberOfParameters());

    Soma alg = new Soma(queue, mqService, mapper, Collections.singletonList(
        commonParameters.getSourceIp()), algorithmEvaluator);

    alg.setPathLength(somaParameters.getPathLength());
    alg.setStep(somaParameters.getStep());
    alg.setPrt(somaParameters.getPrt());

    algorithmRepository.postTask(alg, pop, somaParameters.getNumberOfIterations
        (), AlgorithmType.SOMA);

    return ResponseEntity.ok(parameters);
}
```

Výpis 9: Příklad definice koncového bodu REST API pro spuštění algoritmu SOMA

9 Návrh a implementace generátoru síťového provozu

Generátor síťového provozu využívá vícevláknové zpracování. Implementace je založena na návrhovém vzoru Producer-Consumer. Princip činnosti je znázorněn sekvenčním diagramem na Obrázku 22. Roli producera zajišťuje objekt třídy `WebsocketListener`, jedná se o samostatné vlákno, jehož účelem je navázat Websocket spojení s webovou aplikací. Po úspěšném spojení je vyvolána při každé přijaté zprávě callback metoda `on_message`. Metoda je zodpovědná pouze za vložení přijaté JSON zprávy do fronty pro zpracování. Fronta zpráv je tzv. thread-safe datová kolekce, čímž je zajištěna správná synchronizace činnosti vláken. Další callback metody jako je `on_open` a `on_close` slouží pouze pro výpis informací o stavu spojení, aplikační logiku nezajišťují. Objekt třídy `TrafficHandler` plní roli consumera. Vlákno periodicky kontroluje, zda se ve frontě nachází nová zpráva a pokud ano, započne její zpracování. Prvním krokem je převod JSON objektu na objektovou reprezentaci v Pythonu. Následně je pomocí modulu `PacketFactory` vytvořen síťový provoz, za jehož odeslání je zodpovědný rovněž zmíněný modul.

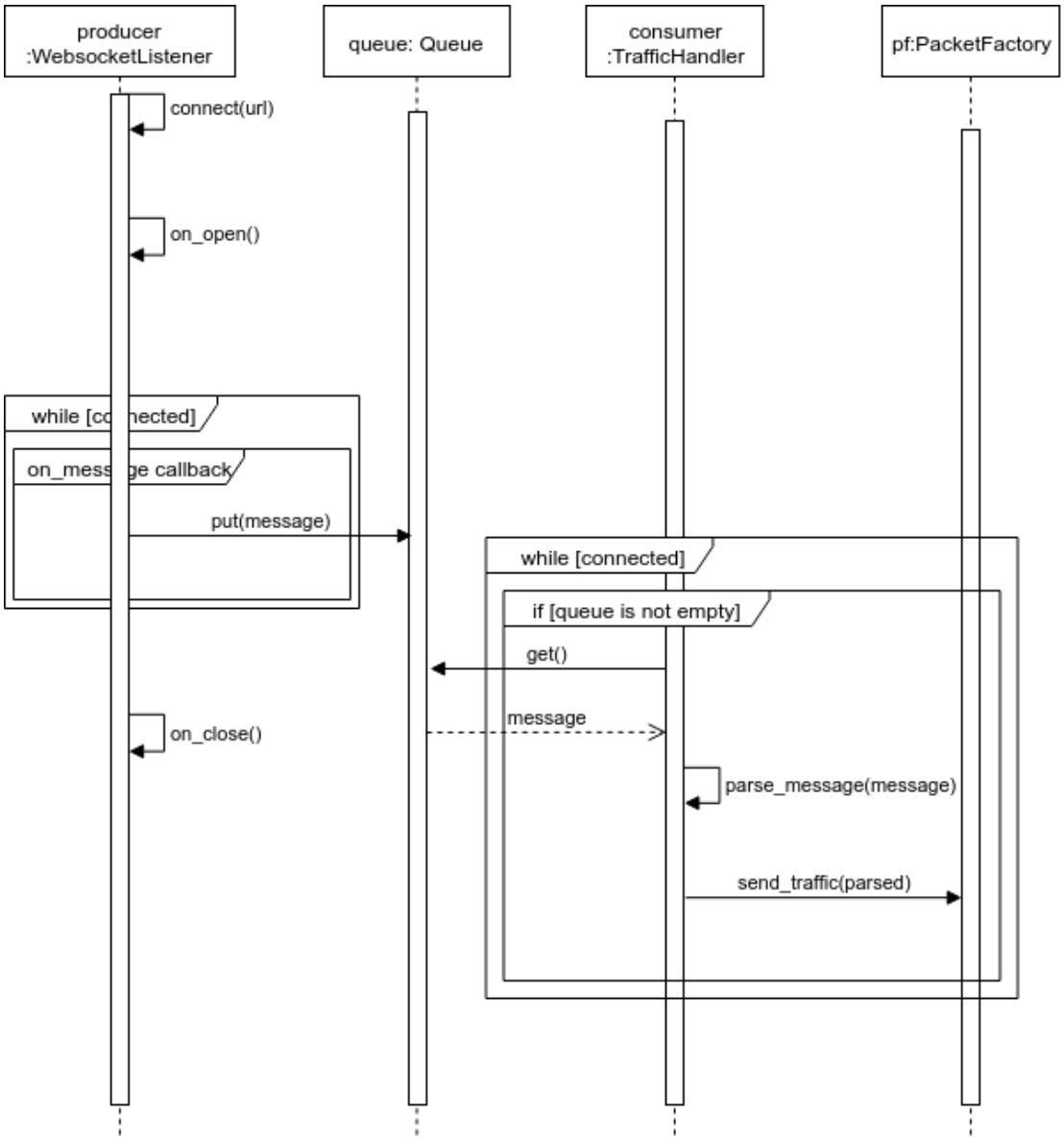
Ve Výpisu 10 vidíme ukázku JSON zprávy, na jejímž základě je vygenerován síťový provoz. Funkce pro vytvoření TCP datagramu je pro ilustraci uvedena ve Výpisu 11. Vidíme, že pole z uvedeného JSON objektu korespondují s jejími parametry. Zbylé položky jako je např. číslo zdrojového portu jsou voleny náhodně. Pro manipulaci se síťovým provozem je využita knihovna Scapy, která dovoluje parametrizaci veškerých polí v hlavičkách protokolů.

```
{
  "trafficPlan": [
    {
      "packetType": "TCP_SYN",
      "ipDst": "192.168.56.10",
      "ipSrc": "192.168.56.21",
      "portDst": 8080,
      "dataSize": 0,
      "winSize": 33718
    },
    ...
  ]
}
```

Výpis 10: Ukázka přenesených informací ve formátu JSON přes Websocket

```
def _tcp_base(ip:str, port:int, flags:str, src_ip:str=None, win:int=512)
    ip = IP(dst=ip) if src_ip == None else IP(dst=ip, src=src_ip)
    return ip/TCP(dport=port,
        sport=random.randint(MIN_EPHEMERAL_PORT, MAX_PORT), window=win,
        flags=flags, seq=random.randint(0, MAX_SEQ_ACK), ack=random.randint(0,
            MAX_SEQ_ACK))
```

Výpis 11: Funkce pro vytvoření datagramu



Obrázek 22: Sekvenční diagram - implementace vzoru Consumer-Producer v generátoru provozu

10 Návrh a implementace uživatelského rozhraní

Uživatelské rozhraní je implementováno jako single-page aplikace postavená na knihovně React. Rozhraní lze logicky rozdělit do dvou vrstev, a to na prezentační a datovou. Následující kapitoly jsou věnovány popisu jejich funkcionality.

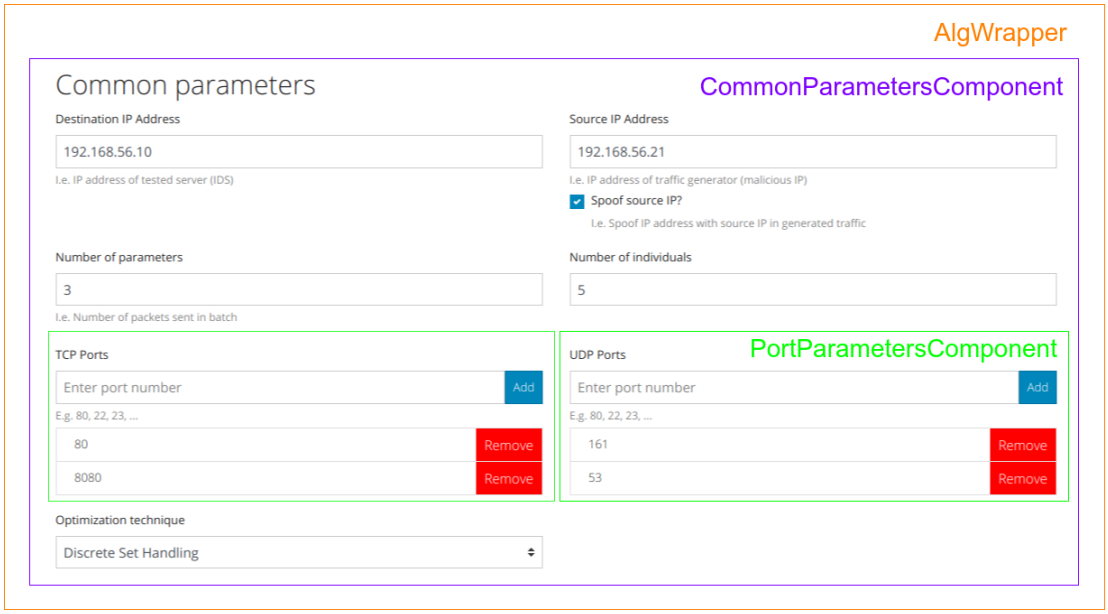
10.1 Komponenty prezentační vrstvy

Ve Výpisu 12 vidíme část komponenty zodpovědné za směrování mezi podstránkami, každá podstránka je definována tagem `Route`. Parametr `path` určuje cestu v URL, která vede k vykreslení komponenty specifikované v parametru `render`. Jádrem je komponenta `AlgWrapper`. Ta obaluje všechny podstránky, důvodem je sjednocení jejich struktury a také vykreslení komponent, které jsou pro všechny podstránky společné. Části, které jsou závislé na zvolené podstránce se předávají do hierarchicky vyšší komponenty `AlgWrapper` formou parametru, jedná se o implementaci vzoru Render-Props. Každý implementovaný algoritmus má vlastní podstránku.

```
export default class Routing extends Component {
  render() {
    return (
        <Switch>
            ...
            <Route path="/soma" exact render={() => (<AlgWrapper
            headerComponent={() => (<SomaHeader />)}
            algComponent={() => (<SomaComponent />)} />)} />
            <Route path="/pso" exact render={() => <AlgWrapper
            headerComponent={() => <PsoHeader/>}
            algComponent={() => <PsoComponent />} />} />
            ....
        </Switch>
    )
  }
}
```

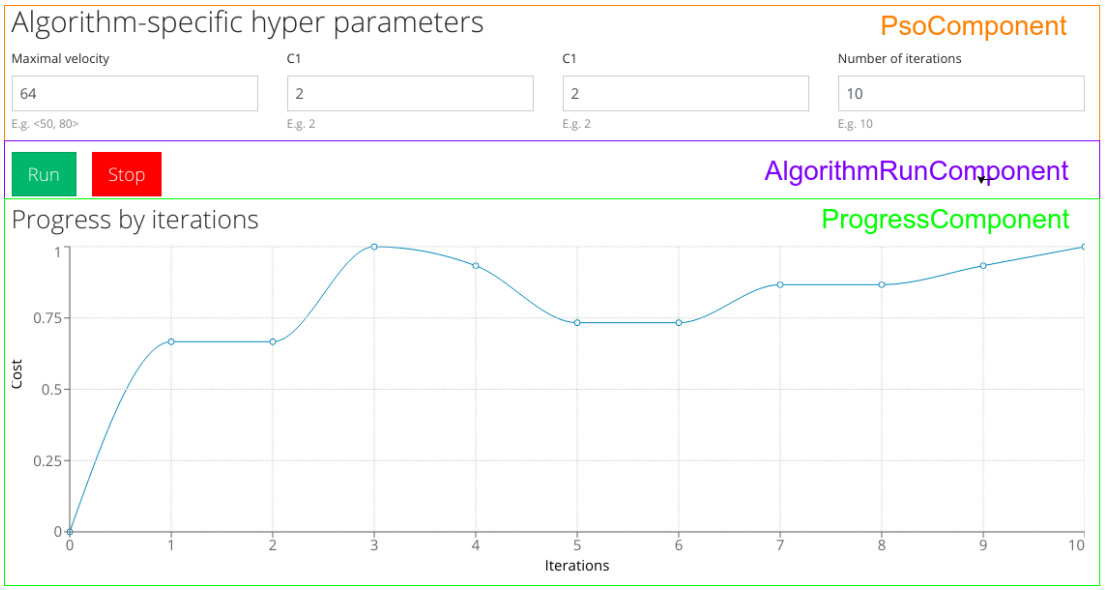
Výpis 12: Ukázka komponenty zajišťující směrování mezi podstránkami

Hierarchii komponent vidíme na Obrázku 23, hlavní komponenty jsou barevně ohraničeny. Komponenta `CommonParametersComponent` zajišťuje část rozhraní pro nastavení parametrů testovacího procesu, které jsou společné pro všechny algoritmy. Nastavení cílových portů je zajištěno samostatnou komponentou `PortParametersComponent`. Nastavené parametry jsou uloženy v datové vrstvě pro aktuální session a není je potřeba nastavovat po přepnutí algoritmu znova.



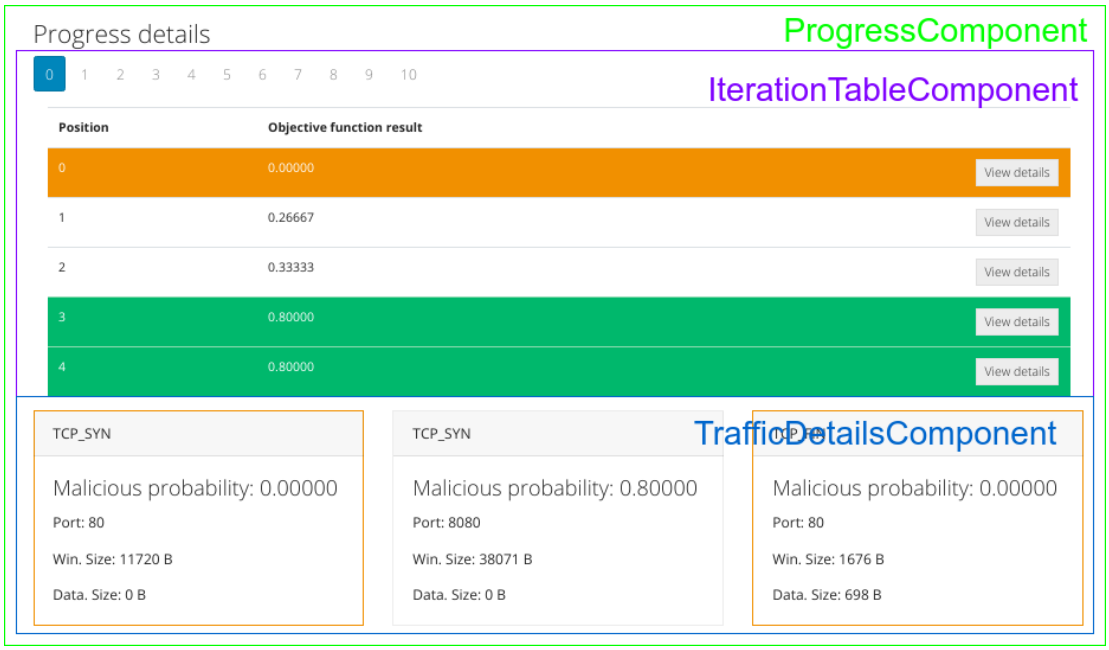
Obrázek 23: Část rozhraní - společné parametry testovacího procesu

Na Obrázku 24 vidíme část rozhraní pro nastavení hyperparametrů zvoleného algoritmu, na ukázce pro algoritmus PSO. Komponenta `AlgorithmRunComponent` je společná pro všechny algoritmy a zajišťuje komunikaci s datovou vrstvou, která je zodpovědná za odeslání HTTP požadavku serverové části penetračního nástroje. Ve spodní části obrázku vidíme také graf, který vizualizuje průběh testovacího procesu. Ten je součástí komponenty `ProgressComponent`, která zajišťuje zobrazení veškerých informací o průběhu, a to ihned, kdy jsou k dispozici v datové vrstvě. Do grafu je vždy vynesena nejnižší hodnota ohodnocení jedince pro danou iteraci.



Obrázek 24: Část rozhraní - parametry algoritmu a vizualizace průběhu

Na Obrázku 25 vidíme další část komponenty `ProgressComponent`. Tabulka v horní části obrázku udává ohodnocení jedinců pro zvolenou iteraci. Z tabulky lze vybrat konkrétního jedince pro zobrazení jeho parametrů, to zajišťuje komponenta `TrafficDetailsComponent`.

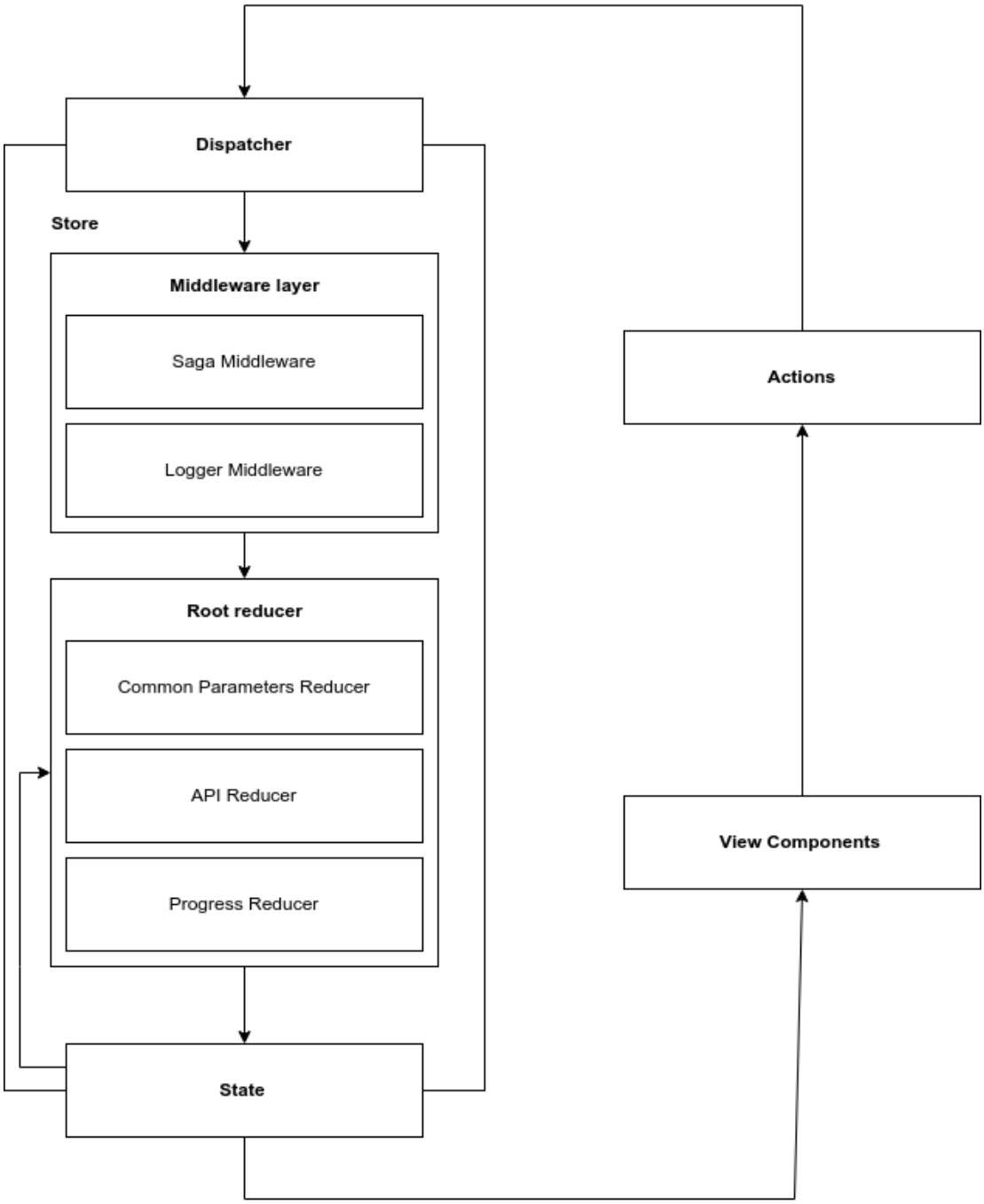


Obrázek 25: Část rozhraní - informace o průběhu

10.2 Datová vrstva

Úkolem datové vrstvy je zprostředkování dat získaných komunikací se serverovou částí aplikace prostřednictvím REST API, resp. protokolu Websocket, komponentám prezentační vrstvy. Moduly zodpovědné za samotnou komunikaci jsou rovněž součástí datové vrstvy.

Implementace datové vrstvy je založena na knihovně Redux, která poskytuje rámec implementující návrhový vzor Flux [81]. Cílem vzoru je oddělit logiku manipulace s daty od prezentační vrstvy, podobně jako je tomu u architektury MVC. Hlavním rozdílem oproti MVC, kromě odlišně struktury komponent, je tok dat pouze v jednom směru. Na Obrázku 26 vidímě, jak je vzor implementován v penetračním nástroji. Základní stavební kameny představují moduly: action, dispatcher, store a reducer. Pokud potřebuje komponenta prezentační vrstvy změnit hodnoty v datové vrstvě, neprovádí změnu přímo, ale prostřednictvím zaslaní zprávy dispatcheru. Zpráva nese informaci o typu prováděné akce a také data, který jsou pro vykonání potřebná. V rámci vzoru Flux reprezentuje zprávu již zmíněná *action*. Konkrétní příklad *action* vidíme ve Výpisu 13, jedná se o požadavek na změnu cílové IP adresy. Úlohou dispatcheru je přijetí těchto dat a jejich předání příslušnému reduceru.



Obrázek 26: Struktura datové vrstvy

```
{
    "type": "UPDATE_DST_IP",
    "ip": "192.168.56.21"
}
```

Výpis 13: Ukázka action objektu pro změnu cílové IP adresy

Dispatcher neobsahuje žádnou aplikační logiku, plní pouze úlohu prostředníka v komunikaci mezi prezentační a datovou vrstvou. Store je kontejner, který obsahuje logiku datové vrstvy a současně slouží jako úložiště stavu aplikace. Ke storu je možné připojit libovolnou komponentu prezentační vrstvy z hierarchie komponent. Výhodou je, že store je v aplikaci jediné místo, kam jsou data ukládána, tudíž nedochází k redundanci a případným nekonzistencím. Než jsou action objekty předány reduceru, projdou middleware vrstvou, která specifikuje operace, které jsou provedeny s každým průchozím objektem. V nástroji se tento postup používá např. pro logování. Reducer je funkce, zajišťující logiku datové vrstvy. Vstupem funkce je aktuální stav dat a action objekt. Na základě příslušné akce se provede transformace aktuálního stavu na stav následující, který je výstupem funkce. Aktuální stav se zásadně nemění, je pouze nahrazován stavem následujícím. V nástroji jsou reducery rozděleny dle oblastí, za které jsou zodpovědné, např. práce s parametry testovacího procesu nebo uchovávání dat o jeho průběhu.

Komunikace se serverovou části nástroje, ať už přes REST API nebo pomocí protokolu Websocket, probíhá asynchronně. Aby nedocházelo ke komplikaci logiky v reducerech, zajišťují komunikaci samostatné moduly. Moduly využívají knihovnu Redux-Saga [80], která implementuje návrhový vzor Saga. Vzor zajišťuje převedení sekvence asynchronních akcí na transakci složenou ze několika synchronních kroků, což řeší problémy souběhu včetně ošetření případných selhání. Implementace v prostředí vyvíjeného nástroje využívá middleware (viz Obrázek 26), který monitoruje stav asynchronních volání REST API a data reducerům předává až ve chvíli kdy jsou dostupná, stejně tak informace o chybách. Tímto způsobem nedochází k problémům souběhu, protože pro reducer se příchozí požadavky již jeví jako synchronní.

```
export function* handlePostAlgorithm(action) {
    try {
        yield put(showSent(null));
        yield put(updateApiState({fieldErrors: []}));
        const url = yield select(getRootUrl);
        yield call(postAlgorithm, action, url);
    } catch (err) {
        yield put(showError(null));
        if (err.status === 400) {
            const body = yield err.json();
            yield put(updateApiState({fieldErrors: body.fieldErrors}));
        }
    }
}
```

Výpis 14: Ukázka asynchronního procesu s využitím vzoru Saga

Ve Výpisu 14 vidíme ukázku funkce, která využívá návrhový vzor Saga pro zpracování asynchronního volání REST API s požadavkem pro spuštění testování. Konstrukce `yield` zajišťuje,

že vykonávání funkce se na asynchronní akci zastaví a pokračuje se až ve chvíli, kdy je akce úspěšně dokončena. Funkce `call` zajišťuje volání funkce pro odeslání požadavku REST API, funkce `put` komunikuje s reducery a předává jim *action* objekty v případě nutnosti změny stavu dat. Komunikace prostřednictvím protokolu Websocket funguje na stejném principu, jediným rozdílem je, že připojení je nutné udržovat aktivní po celou dobu běhu aplikace. Z tohoto důvodu je použita konstrukce `fork`, která umožňuje spuštění samostatného vlákna na pozadí, které spojení udržuje a s reducery komunikuje opět pomocí zmíněné funkce `put`.

10.2.1 Propojení datové a prezentační vrstvy

Komponenty prezentační vrstvy využívají datovou vrstvu výhradně prostřednictvím dispatcheru, samozřejmě je umožnění čtení aktuálního stavu dat ze storu. Ve Výpisu 15 vidíme jak vypadá připojení komponenty `PsoComponent` k datové vrstvě. Figurují zde dvě funkce, a to `mapStateToProps` a `mapDispatchToProps`. První zmíněná funkce předává komponentě aktuální data ze storu. V případě, že dojde k změně dat, komponenta je automaticky vykreslena znova, ale už s aktuálními daty. Druhá zmíněná funkce zpřístupňuje komponentě dispatcher. K dispatcheru komponenta nepřistupuje přímo, ale jsou ji předány reference na funkce, které přístup k němu a vytváření action objektu zapouzdřují. Komponenta je zodpovědná pouze za dodání dat, které jsou v rámci action objektu přeneseny. Samotné připojení k datové vrstvě probíhá voláním funkce `connect`, do které vstupuje dříve zmíněná dvojice funkcí formou referencí.

```
const mapStateToProps = state => ({
  commonParameters: selectCommonParameters(state.commonParametersReducer),
  errors: state.commonParametersReducer.errors
});

const mapDispatchToProps = dispatch => ({
  onSubmit: (common, specific, path) => dispatch(postAlgorithmAction(common,
      specific, path)),
  onCancel: () => dispatch(cancelAlgorithmAction()),
  onUpdateProgress: (progress) => dispatch(updateProgress(progress)),
});

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(PsoComponent);
```

Výpis 15: Ukázka propojení datové vrstvy a komponenty prezentační vrstvy

11 Testování nástroje

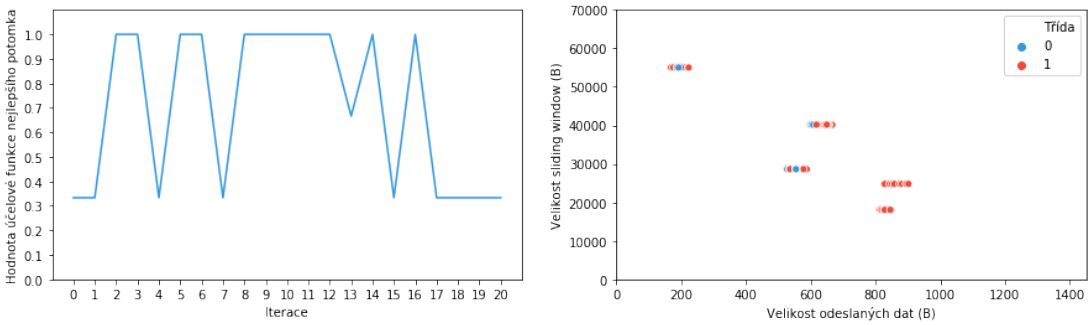
Testování nástroje probíhá ve dvou fázích. První fázi je testování implementovaných algoritmů na IDS, který se v průběhu neučí nová pravidla. Zpětná vazba IDS tedy není závislá na průběhu optimalizačního procesu, jako je tomu v případě učení. Z tohoto důvodu je penalizační faktor ponechán nulový po celou dobu testování. IDS využívá model založený na algoritmu Random forest se třemi rozhodovacími stromy. Model je schopen správně detekovat pouze některé vzory provozu, a to jen pro protokol TCP. Cílem je otestovat implementované algoritmy na zmíněném modelu a analyzovat jejich přístup k hledání optima pro volbu algoritmu, který bude využit ve druhé fázi k učení nového modelu. Za dosažení lokálního optima je považováno nalezení datagramu, který je ohodnocen hodnotou pravděpodobnosti v intervalu $(0,0; 0,8>$. Za globální optimum je považováno nalezení takového datagramu, který bude ohodnocen nulovou hodnotou pravděpodobnosti. Algoritmy byly otestovány s různými konfiguracemi parametrů. Vizualizace zahrnuje vždy spojnicový graf, který vyjadřuje hodnotu účelové funkce nejlepšího potomka v dané iteraci, žádná forma elitismu se zde nevyužívá. Druhý graf je bodový a zobrazuje nalezená řešení ve dvou dimenzích, a to z hlediska velikosti odeslaných dat a velikosti sliding window jednotlivých datagramů. Třída nula je přiřazena těm datagramům, které byly ohodnoceny pravděpodobnosti menší než 0,8, jedna je přiřazena v opačném případě. Druhou fázi je vytvoření nového modelu, který se učí vzory pomocí vybraných algoritmů na základě analýzy z první fáze. V tomto případě je na straně IDS opět využit algoritmus Random forest v rámci klasifikačního modulu. Data všech experimentů jsou součástí elektronické přílohy.

11.1 Reprezentace jedinců založená na parametrech datagramu

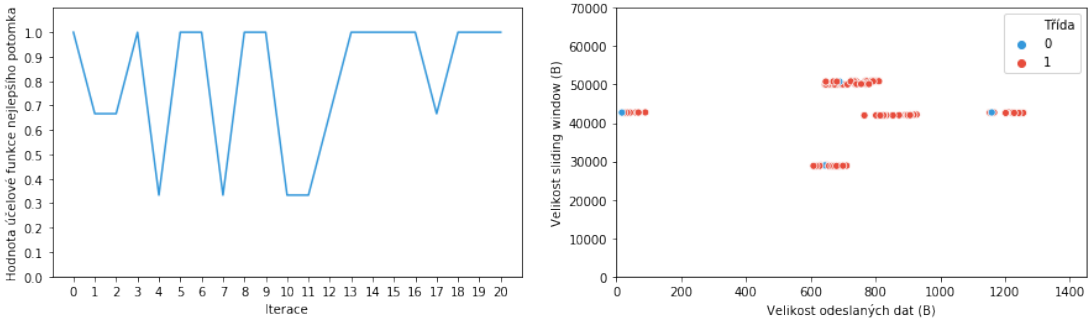
Kapitola se zabývá optimalizací využívající reprezentaci jedinců založenou na parametrech datagramů. Podkapitoly se věnují analýze chování jednotlivých algoritmů. Všechny algoritmy vykonaly 20 iterací, populace se vždy skládala z pěti jedinců.

11.1.1 Simulované žíhání

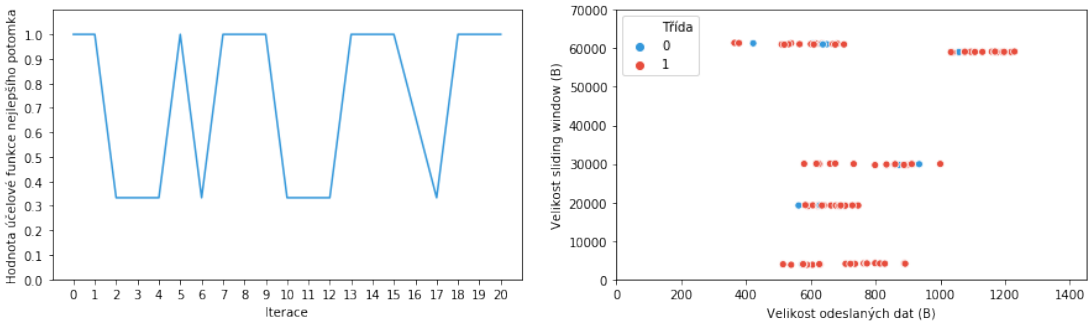
Na Obrázcích 27, 28 a 29 vidíme průběh simulovaného žíhání. Během testování byla měněna směrodatná odchylka. Ze spojnicových grafů vidíme, že algoritmus je schopen dosáhnout pouze lokálního optima, což odpovídá zaměření tohoto algoritmu. Problémem je, že algoritmus prohledává pouze velmi omezené oblasti a nalezená lokální optima jsou navíc často shodná, jak dokazují bodové grafy. Zvětšování směrodatné odchylky vede ke zvětšení prohledávaného prostoru, nicméně algoritmus i tak stále prohledává pouze vybrané oblasti.



Obrázek 27: Simulované žíhání ($T_{start} = 1000; T_{stop} = 0,1; \beta = 0,14; \sigma = 10$)



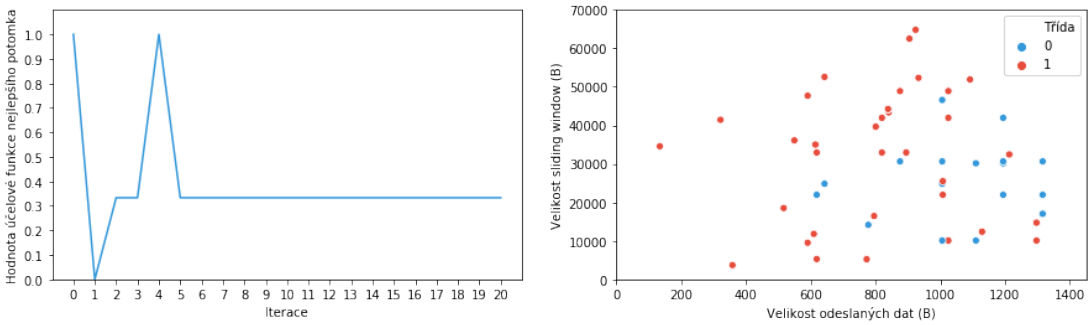
Obrázek 28: Simulované žíhání ($T_{start} = 1000; T_{stop} = 0,1; \beta = 0,14; \sigma = 25$)



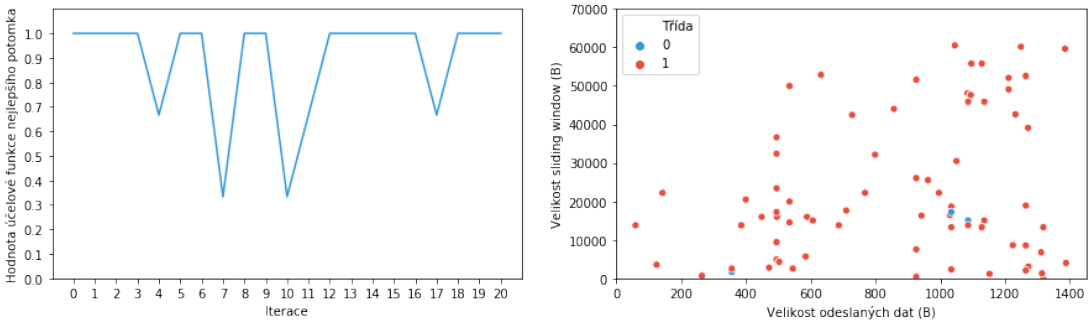
Obrázek 29: Simulované žíhání ($T_{start} = 1000; T_{stop} = 0,1; \beta = 0,14; \sigma = 50$)

11.1.2 Diferenciální evoluce

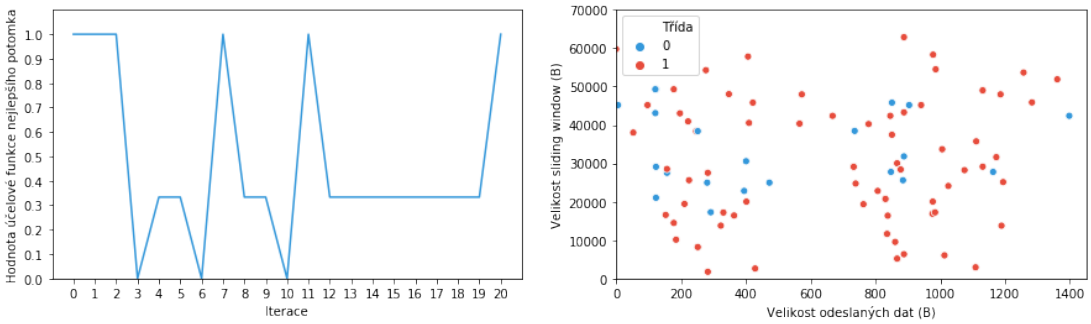
V rámci testování byly měněny strategie mutace. První strategií je Rand-One, jejíž výsledky vidíme na Obrázku 30. Oproti Simulovanému žíhání vidíme, že algoritmus má daleko lepší explorační schopnosti. Výhodou je, že nalezená optima se od sebe liší, jak vidíme na bodových grafech. Algoritmus byl navíc schopen dosáhnout globálního optima, s využitím strategie Current-To-Best dokonce opakovaně, viz Obrázek 32. Strategie Best-One vykazuje horší explorační schopnosti, jak vidíme na bodovém grafu na Obrázku 31, kde je část prostoru naprosto vynechána.



Obrázek 30: Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Rand-One mutace)



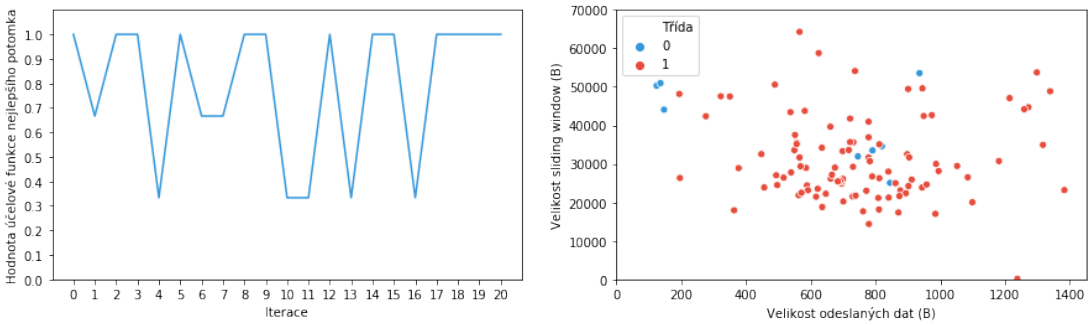
Obrázek 31: Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Best-One mutace)



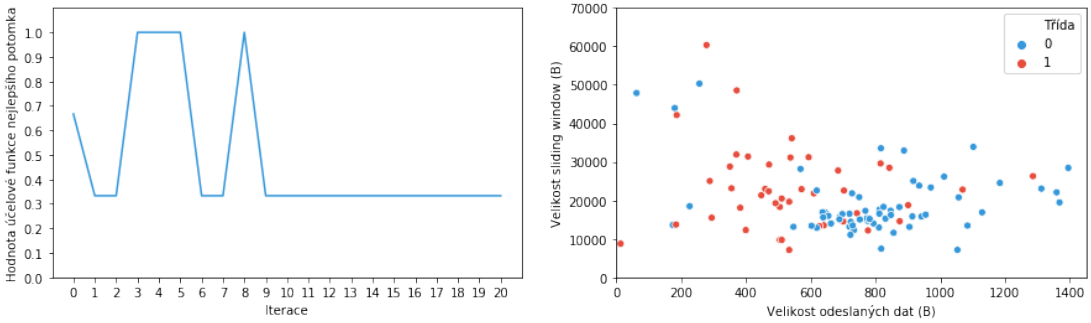
Obrázek 32: Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Current-To-Best mutace)

11.1.3 Grey Wolf Optimization

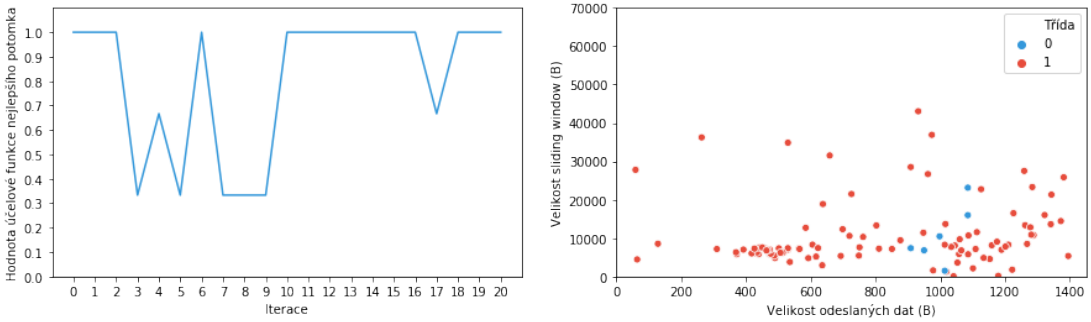
Grey Wolf Optimization nepodporuje nastavení žádných hyperparametrů. Z Obrázků 33, 34 a 35 je patrné, že bylo dosaženo pouze lokálního optima. Vidíme, že algoritmus je zaměřen spíše exploitačně, protože se zaměřuje na prohledávání poměrně omezených oblastí.



Obrázek 33: Grey Wolf Optimization - první spuštění



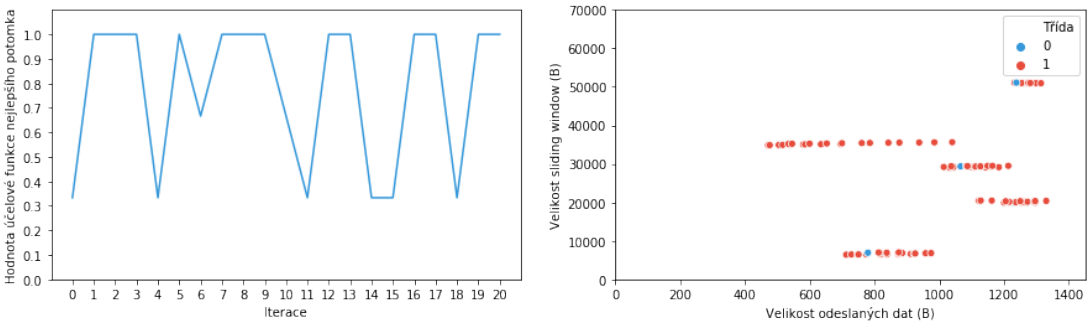
Obrázek 34: Grey Wolf Optimization - druhé spuštění



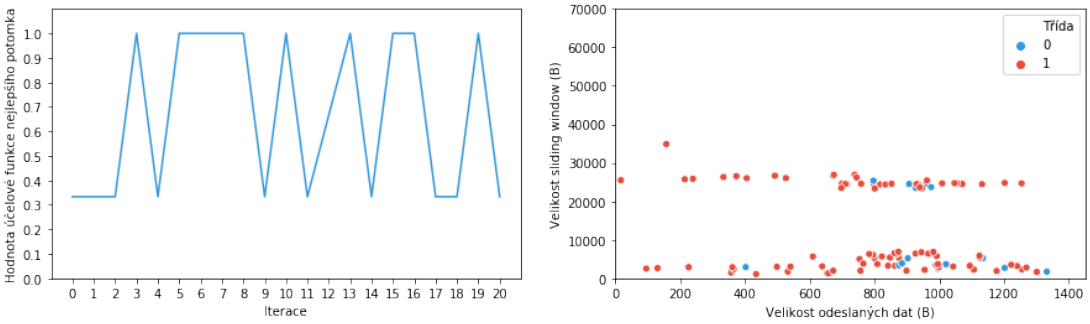
Obrázek 35: Grey Wolf Optimization - třetí spuštění

11.1.4 Particle Swarm Optimization

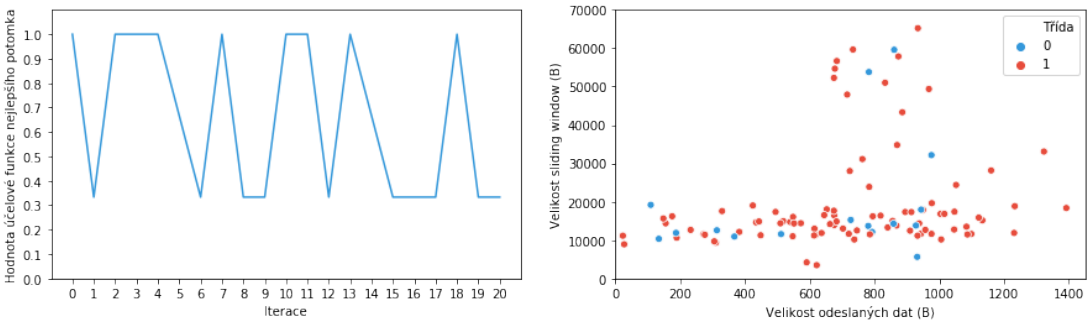
Na Obrázcích 36, 37 a 38 vidíme vliv nastavení maximální rychlosti částice na chování algoritmu. S rostoucí maximální rychlosti rostou i explorační schopnosti algoritmu, pro nás problém je tedy vhodnější volit větší hodnoty. Pokud je maximální rychlost příliš nízká, je algoritmus vhodný pouze k lokální optimalizaci. Globální optimum nebylo nalezeno ani v jednom případě, byla dosažena pouze lokální optima.



Obrázek 36: Particle Swarm Optimization ($V_{max} = 64; C_1 = 2, C_2 = 2$)



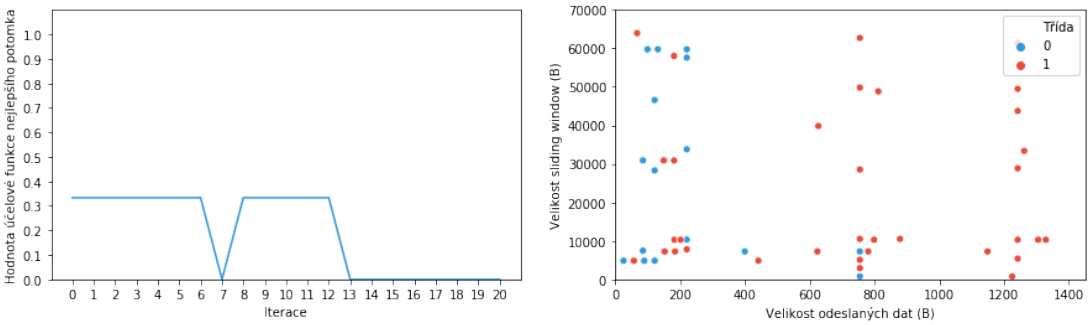
Obrázek 37: Particle Swarm Optimization ($V_{max} = 256; C_1 = 2, C_2 = 2$)



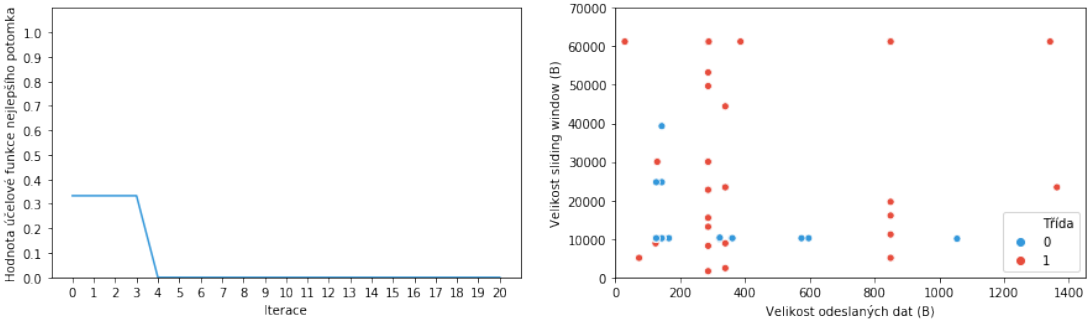
Obrázek 38: Particle Swarm Optimization ($V_{max} = 512; C_1 = 2, C_2 = 2$)

11.1.5 SOMA

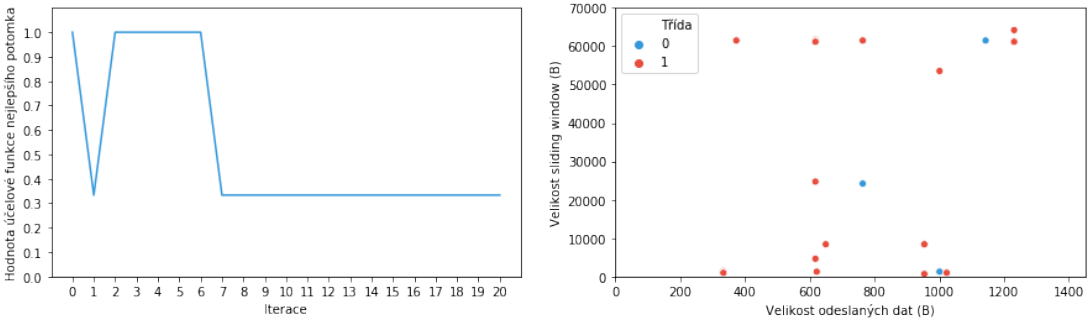
Testování algoritmu SOMA spočívalo ve změně délky skoku jedince. Z grafů na Obrázcích 39, 40 a 41 vidíme, že algoritmus dosahuje lepších výsledků, pokud je délka skoku menší a prostor je tak prohledáván detailněji. Ve dvou ze tří případu nalezl algoritmus globální optimum, s větší délkou skoku bylo nalezeno pouze lokální. Z bodových grafů vidíme, že nejdříve s projevuje explorační charakter algoritmu a po nalezení optima je detailně prohledáno jeho okolí.



Obrázek 39: SOMA (Path = 3; Step = 0,9; PRT = 0,14)



Obrázek 40: SOMA (Path = 3; Step = 1,1; PRT = 0,14)



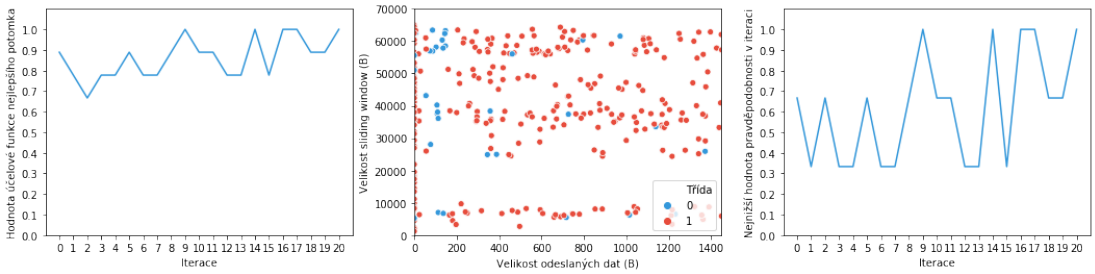
Obrázek 41: SOMA (Path = 3; Step = 1,6; PRT = 0,14)

11.2 Reprezentace jedinců využívající discrete set handling

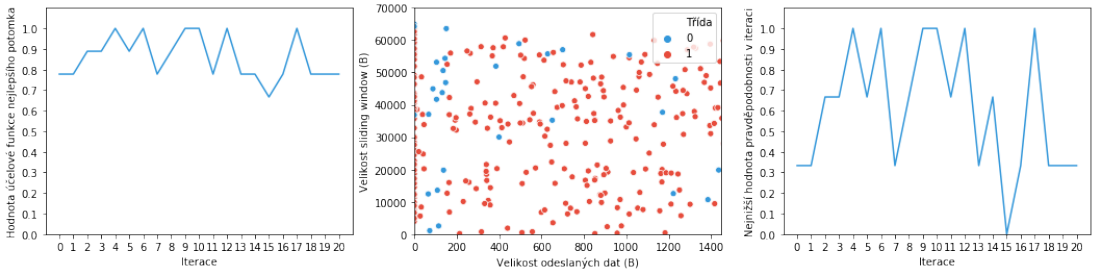
Testované algoritmy opět vykonaly 20 iterací a populace se skládaly z pěti jedinců. Rozdílný je zde počet ohodnocení, protože každý jedinec reprezentuje trojici datagramů. Vizualizace výsledků odpovídá předchozí kapitole, nicméně přibyl další spojnicový graf, který zobrazuje nejnižší hodnotu ohodnoceného datagramu v každé iteraci. Celkové ohodnocení jedince je dáno vztahem, jenž je uveden v kapitole 8.3 a na základě tohoto grafu je možné porovnávat nalezená řešení na úrovni datagramů napříč reprezentacemi.

11.2.1 Simulované žíhání

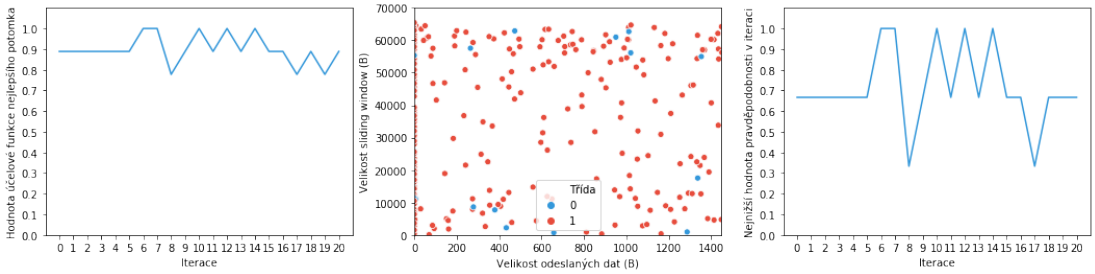
Na Obrázcích 42, 43 a 44 vidíme průběh simulovaného žíhání pro druhou reprezentaci jedinců. Z bodových grafů vidíme, že díky složení jednoho jedince z více datagramů je prostor prohledán mnohem detailněji, než tomu bylo u předchozí reprezentace. Globálního optima bylo dosaženo pouze jednou, v ostatních případech byla nalezena pouze lokální optima. Vlivem reprezentace jedinců došlo k značnému potlačení exploitačních schopností algoritmu a nalezená řešení jsou více rozptýlená v prostoru. Nicméně i tak vidíme, že při nižších hodnotách směrodatné odchylky se nalezená řešení shlukují. Nejvíce patrný je tento jev na Obrázku 42, kde vidíme výrazné shluky bodů.



Obrázek 42: Simulované žíhání ($T_{start} = 1000; T_{stop} = 0,1; \beta = 0,14; \sigma = 10$)



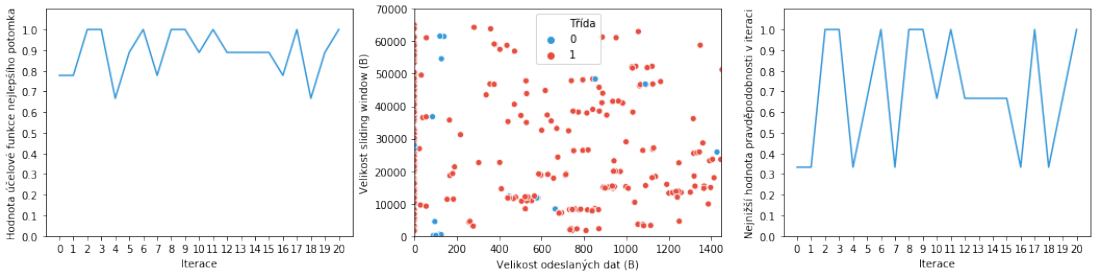
Obrázek 43: Simulované žíhání ($T_{start} = 1000; T_{stop} = 0,1; \beta = 0,14; \sigma = 25$)



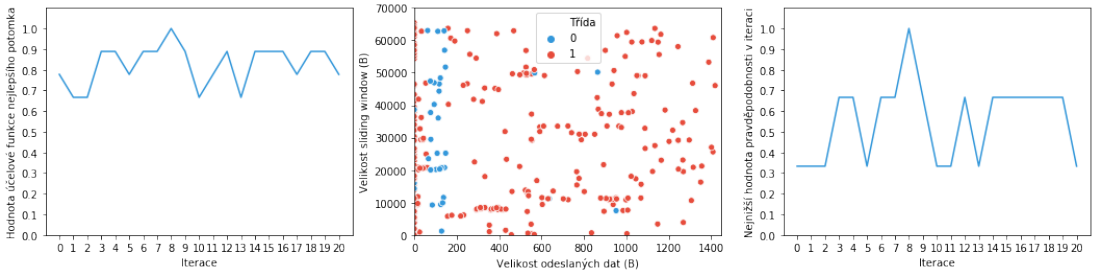
Obrázek 44: Simulované žíhání ($T_{start} = 1000; T_{stop} = 0,1; \beta = 0,14; \sigma = 50$)

11.2.2 Diferenciální evoluce

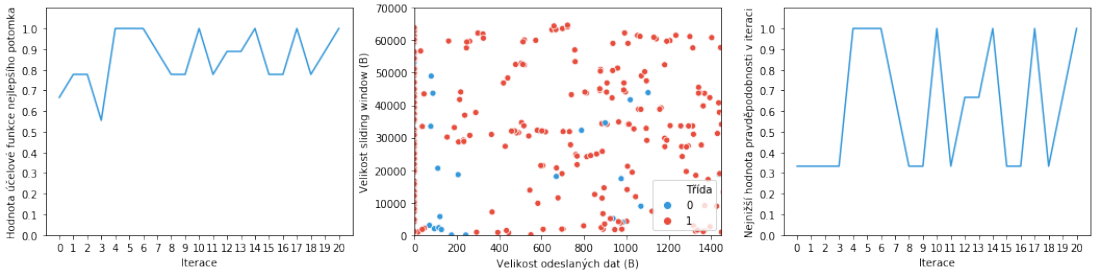
Explorační schopnosti algoritmu jsou závislé na zvolené strategii mutace. Na Obrázku 46 vidíme výsledky pro strategii Best-One, ze kterých je patrné, že po nalezení lokálního optima se algoritmus zaměřil na prohledávání jeho okolí. Tím došlo k objevení většího množství lokálních optim, než je tomu u zbylých dvou strategií. Nevýhodou je, že nalezená optima jsou soustředěná v jedné oblasti na rozdíl od ostatních strategií, jak můžeme vidět na Obrázcích 45 a 47. Globálního optima nebylo dosaženo v žádném z experimentů.



Obrázek 45: Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Rand-One mutace)



Obrázek 46: Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Best-One mutace)

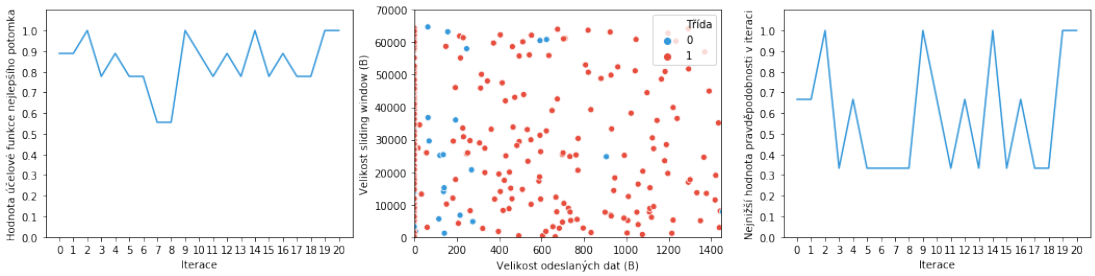


Obrázek 47: Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Current-To-Best mutace)

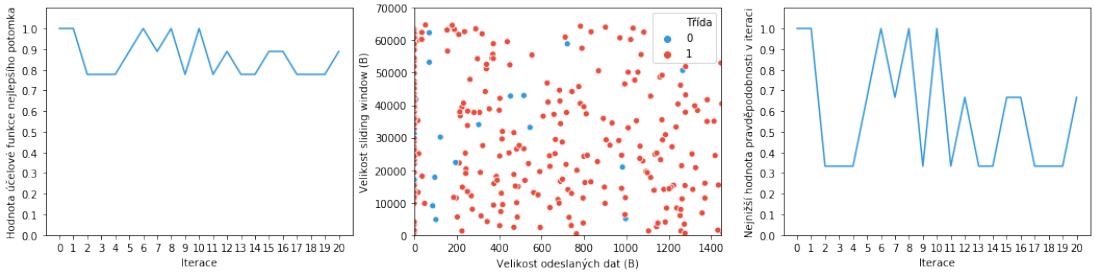
11.2.3 Grey Wolf Optimization

Absence hyperparametrů způsobuje, že nejsme schopni ovlivnit chování algoritmu, který je plně závislý na počáteční populaci jedinců. Na Obrázcích 48 a 49 vidíme, že algoritmus byl schopen

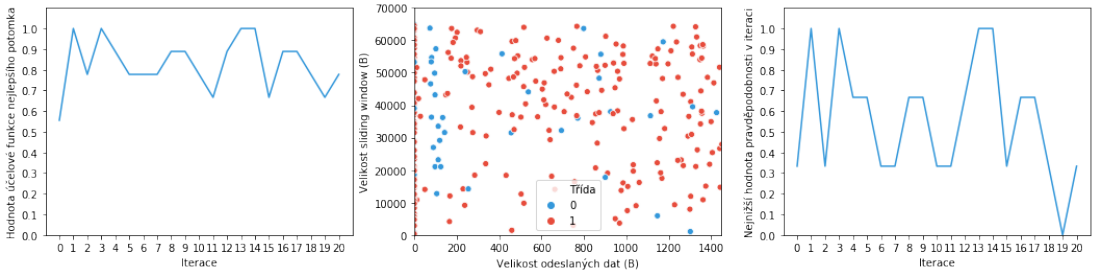
nalézt různá lokální optima a v žádné oblasti neuvízl. Oproti tomu při třetím spuštění nalezená řešení v zásadě odpovídají řešením nalezeným pomocí Diferenciální evoluce se strategií mutace Best-One. Tento jev dokazují Obrázky 50 a 46, kdy oba algoritmy detailněji prohledávaly totožnou oblast prostoru možných řešení.



Obrázek 48: Grey Wolf Optimization - první spuštění



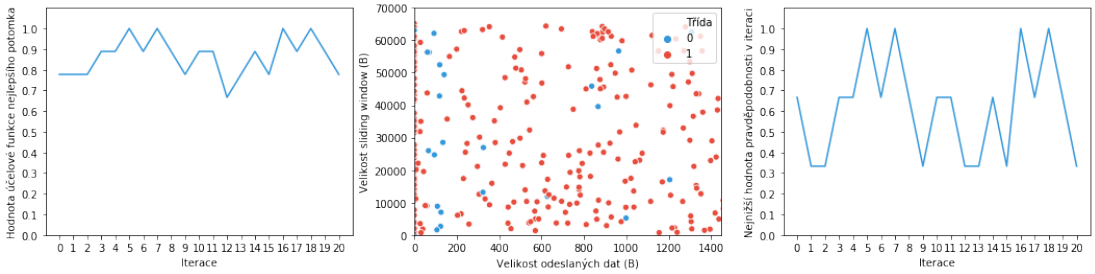
Obrázek 49: Grey Wolf Optimization - druhé spuštění



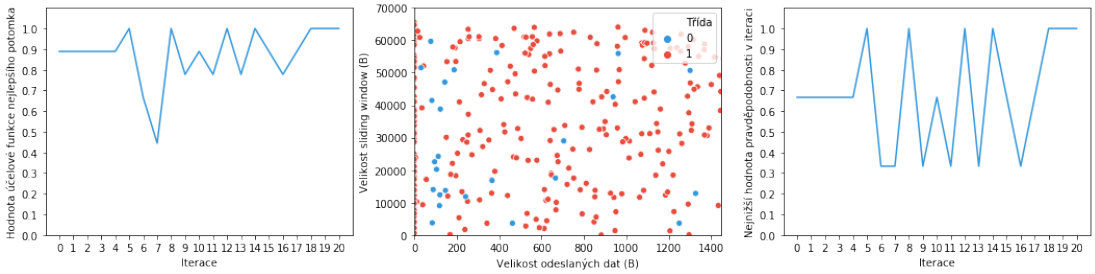
Obrázek 50: Grey Wolf Optimization - třetí spuštění

11.2.4 Particle Swarm Optimization

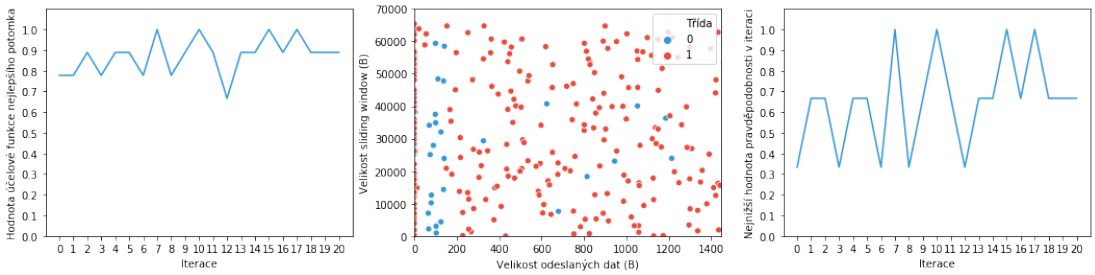
Algoritmus fungoval lépe s vyššími hodnotami maximální rychlosti. Ve všech případech došlo prohledávání ke stejné oblasti, nicméně při vyšších hodnotách V_{max} byla oblast prohledána rovnoměrněji, než tomu bylo při nižší hodnotě. Na Obrázku 51 vidíme, že nalezená řešení mají více shlukovitou strukturu než je tomu u vyšších hodnot V_{max}, což dokazují grafy na Obrázcích 52 a 53.



Obrázek 51: Particle Swarm Optimization ($V_{max} = 64; C_1 = 2, C_2 = 2$)



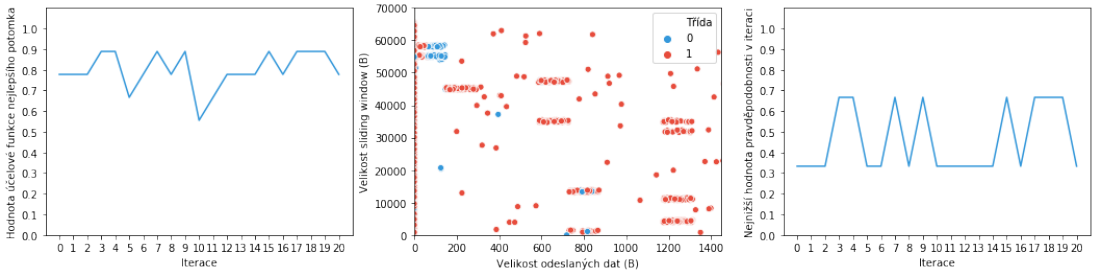
Obrázek 52: Particle Swarm Optimization ($V_{max} = 256; C_1 = 2, C_2 = 2$)



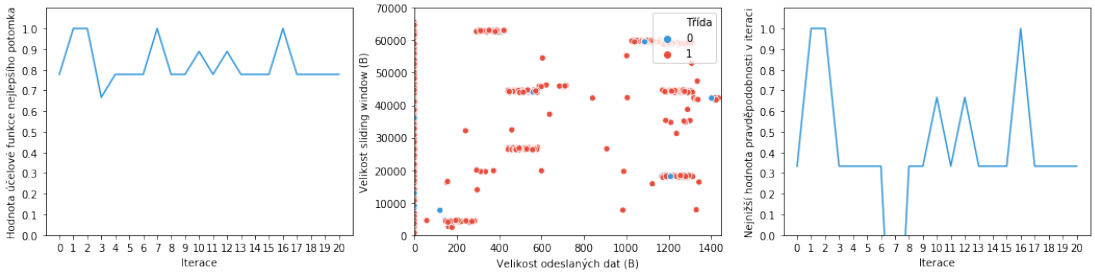
Obrázek 53: Particle Swarm Optimization ($V_{max} = 512; C_1 = 2, C_2 = 2$)

11.2.5 SOMA

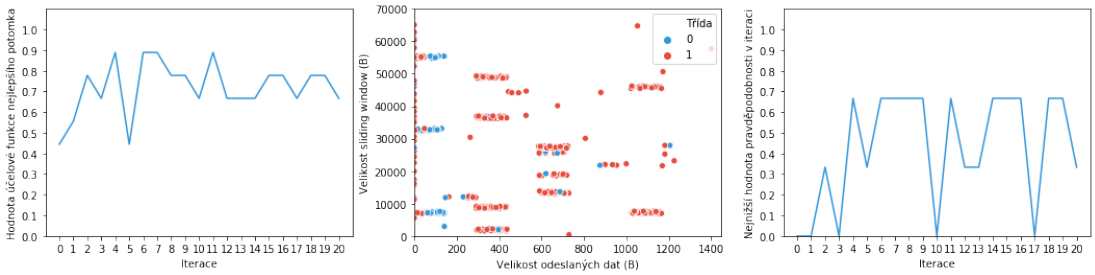
Prohledávání prostoru řešení algoritmem SOMA se naprosto vymyká přístupům jiných algoritmů. Na Obrázcích 54, 55 a 56 vidíme, že prohledávání prostoru je opět rozděleno specifických oblastí, podobně jako jsme mohli pozorovat u předchozí reprezentace jedinců (kapitola 11.1.5). Algoritmus pracuje lépe při vyšších hodnotách délky skoku, na Obrázku 56 vidíme, že algoritmus byl schopen opakovaně nalézt datagramy, které jsou ohodnoceny nulovou hodnotou pravděpodobnosti. Takové chování nevykazoval žádný jiný z testovaných algoritmů.



Obrázek 54: SOMA (Path = 3; Step = 0,9; PRT = 0,14)



Obrázek 55: SOMA (Path = 3; Step = 1,1; PRT = 0,14)



Obrázek 56: SOMA (Path = 3; Step = 1,6; PRT = 0,14)

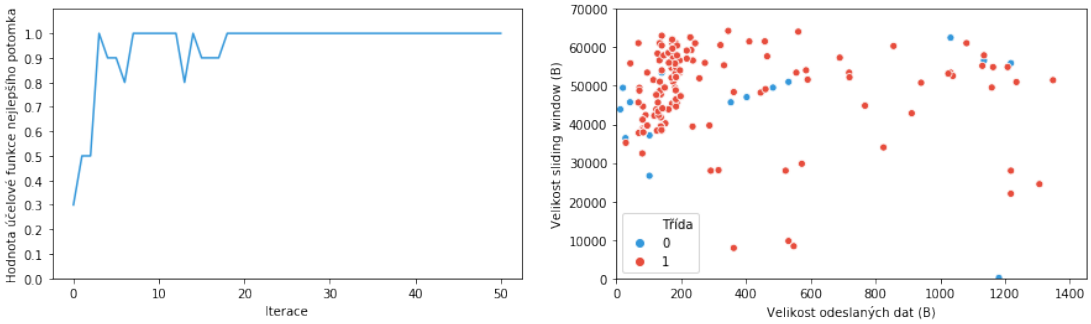
11.3 Učení nového klasifikačního modelu IDS

Na základě testování algoritmů v předchozí kapitole byla vybraná trojice algoritmů, které jsou využity pro učení nových klasifikačních modelů. Pro reprezentaci jedinců založenou na parametrech datagramů byly vybrány algoritmy SOMA a Diferenciální evoluce, protože oba algoritmy dokázaly opakovaně nalézt globální optima. Pro reprezentaci využívající discrete set handling byl vybrán pouze algoritmus SOMA, který byl jako jediný schopen nalézt řešení, která představovala globální optima. Učení probíhá vždy po 50 iterací. Každý takto vytvořený model je následně podroben testování pomocí algoritmů SOMA a Diferenciální evoluce po 30 iterací, a to s využitím obou reprezentací jedinců. Poslední fází testování je generování provozu nástrojem Hping3 (viz kapitola 4.3) pro ověření, že model je schopen správně klasifikovat provoz, který je

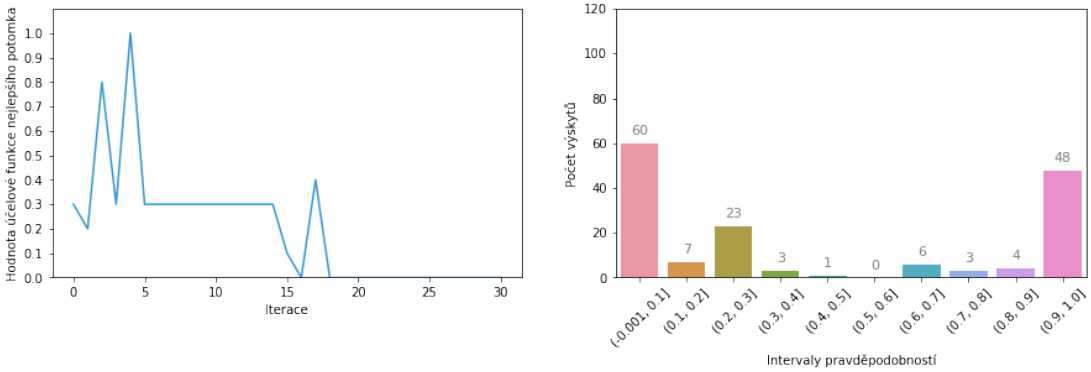
vytvořen pomocí běžně používaných nástrojů. Za správnou klasifikaci datagramu je považováno ohodnocení vyšší hodnotou pravděpodobnosti než 0,8.

11.3.1 Diferenciální evoluce

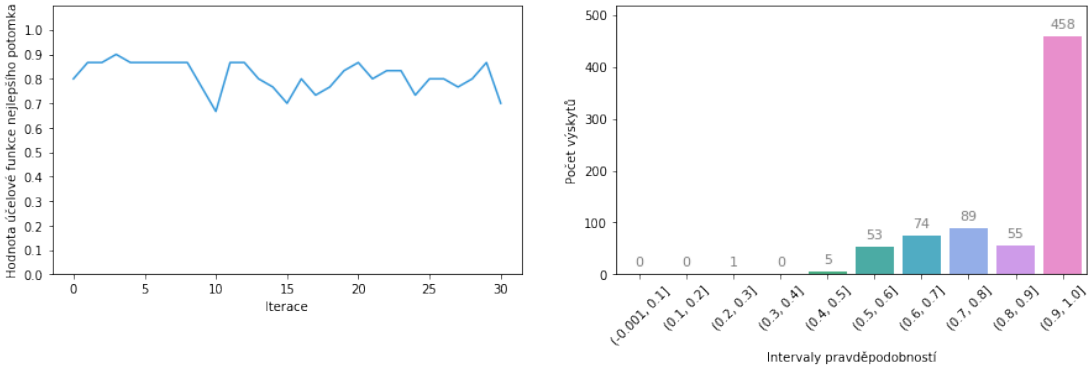
Na Obrázku 57 vidíme průběh učícího procesu pomocí algoritmu Diferenciální evoluce, která využívá reprezentaci jedinců založenou na parametrech datagramů. Z bodového grafu lze usoudit, že se algoritmus zaměřil na průzkum pouze omezené oblasti, což naznačuje, že výsledný model pravděpodobně nebude kvalitní z hlediska generalizace. Grafy na Obrázcích 58 a 59 tento předpoklad potvrzují. Vidíme, že pomocí diferenciální evoluce bylo nalezeno 103 datagramů, které byly ohodnoceny pravděpodobností nižší než 0,8. Pouze 52 datagramů bylo tedy klasifikováno správně.



Obrázek 57: Učení pomocí Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Current-To-Best mutace)



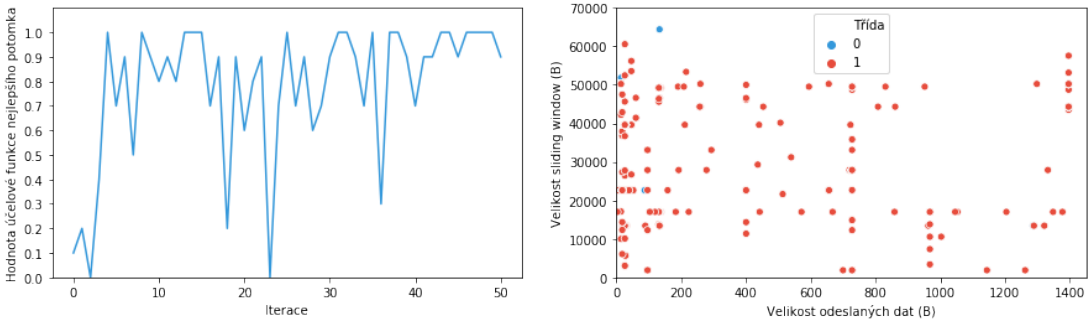
Obrázek 58: Testování pomocí Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Current-To-Best mutace)



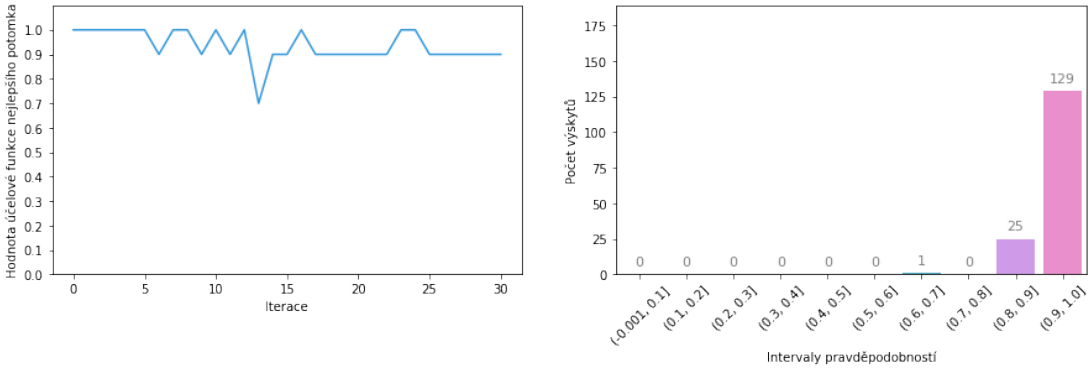
Obrázek 59: Testování pomocí algoritmu SOMA (DSH) (Path = 3; Step = 1,6; PRT = 0,14)

11.3.2 SOMA

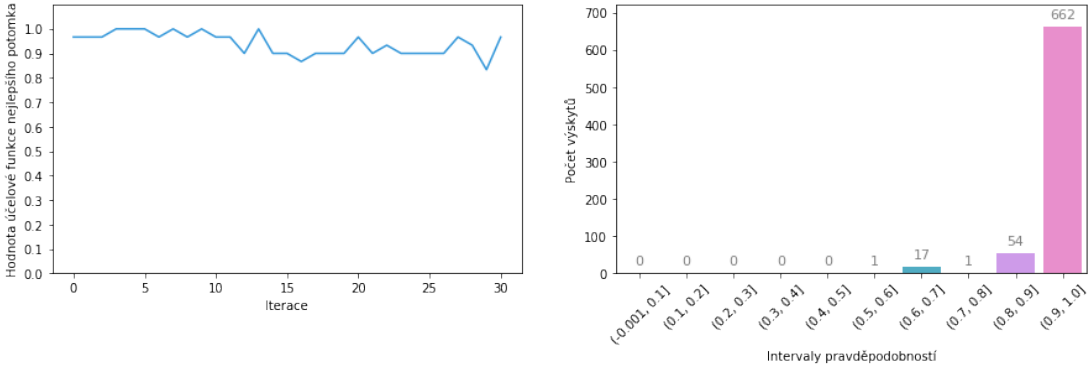
Druhý vytvořený model opět využívá reprezentaci jedinců založenou na parametrech datagramů. Pro učení byl tentokrát použit algoritmus SOMA. Z grafů na Obrázku 60 vidíme, že algoritmus byl schopen optimalizovat dynamickou účelovou funkci mnohem lépe než diferenciální evoluce, která začala stagnovat brzy po dvacáté iteraci. Bodový graf navíc dokazuje, že algoritmus má i lepší explorační schopnosti. Diferenciální evoluci byl nalezen pouze jeden datagram, který byl klasifikován chybně. SOMA nalezla 19 chybně klasifikovaných datagramů, nicméně jejich pravděpodobnosti se pohybovaly v intervalu $(0,6; 0,7>$. Výsledný model je velmi robustní, jak dokazují i grafy na Obrázcích 61 a 62.



Obrázek 60: Učení pomocí alg. SOMA (Path = 3; Step = 0,9; PRT = 0,14)



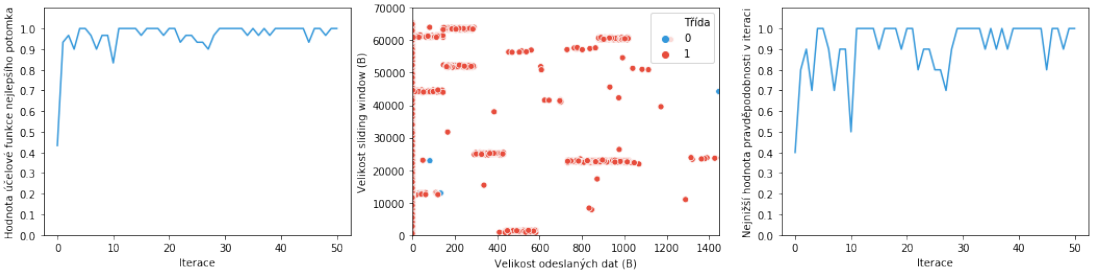
Obrázek 61: Testování pomocí Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Current-To-Best mutace)



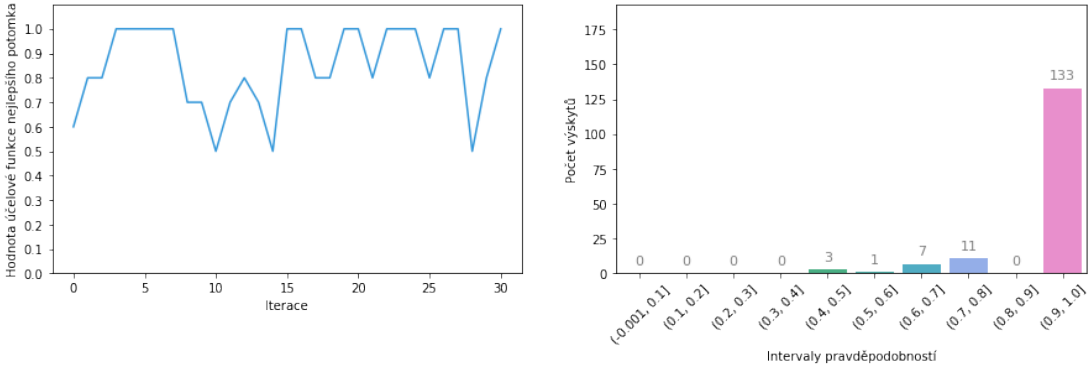
Obrázek 62: Testování pomocí algoritmu SOMA (DSH) (Path = 3; Step = 1,6; PRT = 0,14)

11.3.3 SOMA - reprezentace jedinců využívající discrete set handling

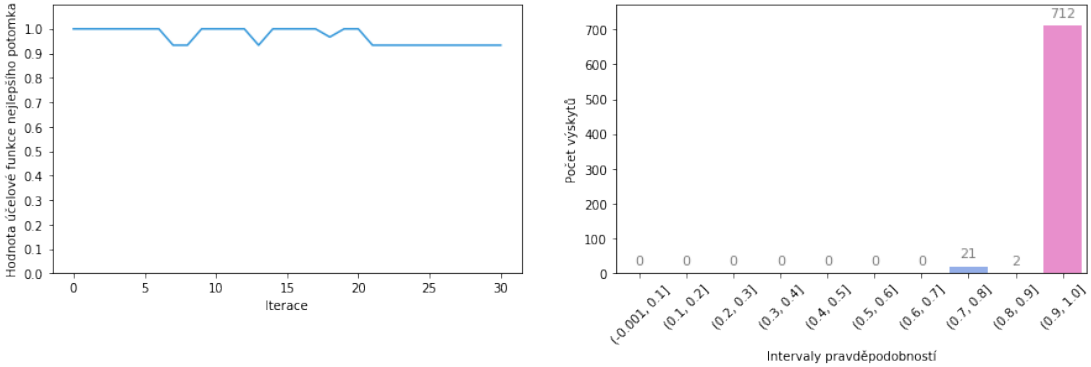
Poslední vytvořený model využívá reprezentaci jedinců založenou na principu discrete set handling a také byl při jeho učení využit algoritmus SOMA. Způsob prohledávání prostoru možných řešení vykazuje podobné charakteristiky jak na dynamické účelové funkci, tak na statické. Jev je patrný z bodových grafů na Obrázcích 63 a 56. Vytvořený model je, stejně jako předchozí, velmi robustní a také výrazně předčil model vytvořený s využitím Diferenciální evoluce. Na Obrázcích 64 a 65 vidíme výsledky testování. Diferenciální evoluce nalezla 22 datagramů, které byly klasifikovány špatně, z celkových 155 ohodnocení. SOMA jich odhalila celkem 21, zbylých 714 datagramů bylo klasifikováno správně.



Obrázek 63: Učení pomocí alg. SOMA (Path = 3; Step = 1,6; PRT = 0,14)



Obrázek 64: Testování pomocí Diferenciální evoluce (F = 0,8; CR = 0,9; Binomické křížení, Current-To-Best mutace)



Obrázek 65: Testování pomocí algoritmu SOMA (DSH) (Path = 3; Step = 1,6; PRT = 0,14)

11.3.4 Testování pomocí nástroje Hping3

Poslední fází testování bylo otestování vytvořených modelů pomocí nástroje Hping3. Každý model byl testován pomocí sekvence 403 datagramů, které si lišily objemem zaslaných dat, příznaky a velikosti sliding window. Rozdíl mezi sousedními hodnotami velikosti dat a sliding window je konstantní, prohledávání prostoru možných řešení je tedy realizováno metodou grid search. Výsledky vidíme v tabulce 7. Všechny modely byly schopny správně klasifikovat většinu

zaslaných datagramů. Stejně jako předešlé testy i výsledky tohoto testování vypovídají o tom, že model vytvořený s využitím Diferenciální evoluce byl schopen klasifikovat provoz hůře než tomu bylo u zbylých dvou modelů, kde byl využit alg. SOMA.

Tabulka 7: Výsledky testování pomocí nástroje Hping3

Interval \Model	DE	SOMA (DSH)	SOMA
$[0.0, 0.1]$	0	0	0
$(0.1, 0.2]$	0	0	0
$(0.3, 0.4]$	0	0	0
$(0.2, 0.3]$	1	0	1
$(0.4, 0.5]$	1	0	0
$(0.7, 0.8]$	4	1	0
$(0.8, 0.9]$	6	2	0
$(0.6, 0.7]$	11	3	26
$(0.5, 0.6]$	13	0	2
$(0.9, 1.0]$	369	397	374

12 Závěr

Práce se zabývá návrhem a implementací automatického penetračního nástroje, který cílí na problematiku testování adaptivních systémů pro detekci průniku.

Práce je rozdělená do dvou částí. První část se zabývá jak procesem penetračního testování, z pohledu jednotlivých fází a přístupů k jejich realizaci, tak možnostmi často používaných nástrojů, které umožňuji část činností automatizovat. Prostor není věnován pouze běžně používaným nástrojům, ale část práce se věnuje také vývoji v oblasti využití umělé inteligence v rámci penetračního testování, která je v dnešních nástrojích zastoupena především strojovým učením. Součástí první části je také detailní srovnání nástrojů určených pro generování síťového provozu, konkrétně se jedná o nástroje Nmap, Hping a Masscan. Závěr první části je věnován oblasti evolučních výpočetních technik a strojovému učení, a to jak obecným principům, tak detailnímu popisu vybraných algoritmů z těchto oblastí.

Začátek druhé části se věnuje analýze datové sady UNSW-NB15 [76]. Výstup analýzy je základem návrhu a implementace vlastního IDS, který je založen na strojovém učení a představuje nedílnou součást testovacího prostředí, které je rovněž představeno v úvodu druhé části. Následující kapitoly se věnují návrhu a implementaci komponent samotného penetračního nástroje, který pro testování IDS využívá evoluční výpočetní techniky. Tato část se nezaměřuje čistě na vývoj nástroje z pohledu softwarového inženýrství, ale současně se věnuje také reprezentaci jedinců a s tím spojenému návrhu účelové funkce. Závěr druhé části se zabývá testování vyvinutého řešení s využitím již zmíněného IDS. Závěrečná fáze zahrnuje jak testování statických klasifikačních modelů, tak učení adaptivních variant.

Problémy v rámci vývoje byly především implementačního charakteru a týkaly se zpravidla komunikace mezi komponentami v reálném čase a s tím spojených problémů souběhu více vláken. Veškerá úskalí se podařilo vyřešit a dle mého názoru nástroj splnil cíle stanovené na počátku vývoje. Nástroj je navržen tak, aby mohl být v budoucnu snadno rozšířen o novou funkcionalitu. Za nejužitečnější rozšíření považuji přidání dalších typů útoků a také nových optimalizačních algoritmů. V rámci implementovaného IDS by, dle mého názoru, byla nejzásadnějším rozšířením možnost integrace s již existujícími firewally v podobě transformace klasifikačního modelu do podoby pravidel, se kterými firewally pracují.

Z výsledků testování nástroje vyplývá, že kvalita nalezených řešení není ovlivněna jen tím, jaký je zvolen optimalizační algoritmus, ale významnou roli hraje také přístup k reprezentaci jedinců a v neposlední řadě návrh účelové funkce. Nejvýrazněji se zmíněné faktory projevily v rámci optimalizace pomocí Simulovaného žíhání, kdy se při přímé optimalizaci parametrů datagramů projevily silné exploitační schopnosti algoritmu, zatímco přechod k reprezentaci jedinců pomocí kolekce vzorů datagramů způsobil výrazné zlepšení jeho exploračních schopností. Učení nových klasifikačních modelů ukázalo, že dosáhnutí velmi dobrých výsledků při testování statického klasifikačního modelu nutně nezaručuje stejně dobré výsledky i v případě použití pro učení nových klasifikačních modelů, jejichž účelová funkce je časově závislá a v průběhu

testovacího procesu se tedy mění. Jev dokládá např. Diferenciální evoluce, kdy při testování statického klasifikačního modelu dosahoval algoritmus lepších výsledků než mnohé další algoritmy, nicméně při optimalizaci dynamické účelové funkce naprosto selhal a vytvořený model byl nejhorší z trojice vytvořených modelů. Po ukončení učící fáze byl vytvořený model opět otestován pomocí Diferenciální evoluce a algoritmus si na, nyní již statické účelové funkci, vedl opět skvěle. Pro reprezentaci jedinců využívající kolekci vzorů datagramů si vedl nejlépe algoritmus SOMA a navzdory tomu, že v testování statických modelů nedosáhl stejných kvalit jako při reprezentaci jedinců založené na parametrech datagramu, tak jeho využití v učící fázi vedlo k vytvoření velmi robustního modelu. Testování modelů pomocí nástroje Hping3 metodou grid search ukázalo, jak velký přínos má využití evolučních výpočetních technik pro oblast testování IDS, kdy ani s více než dvojnásobným počtem ohodnocení oproti např. již zmíněné Diferenciální evoluci, nebylo nalezeno tolik chyb modelu, jako při využití evolučních algoritmů.

Diplomová práce pro mne byla obrovským přínosem, protože jsem si díky ní rozšířil obzory nejen v oblasti informační bezpečnosti, ale zejména jsem nabyl cenné zkušenosti s aplikací evolučních výpočetních technik a strojového učení pro řešení reálných problémů v dnešních systémech.

Literatura

[1] SOBERS, Rob. *60 Must-Know Cybersecurity Statistics for 2019* [online]. [cit. 2019-02-15]. Dostupné z: https://www.varonis.com/blog/cybersecurity-statistics/

[2] *Ten cyber security facts and statistics for 2018* [online]. [cit. 2019-02-15]. Dostupné z: https://us.norton.com/internetsecurity-emerging-threats-10-facts-about-todays-cybersecurity-landscape-that-html

[3] BALOCH, Rafay. *Ethical Hacking and Penetration Testing Guide*. CRC Press, 2015. ISBN 978-1-4822-3162-5.

[4] WEIDMAN, Georgia. *Penetration testing - A Hands-On Introduction to Hacking.* 2014. ISBN 978-1-59327-564-8.

[5] ERICKSON, Jon. *Hacking - The art of exploitation.* 2nd ed. San Francisco: No Starch Press, 2008. ISBN 978-1-59327-144-2.

[6] H. M. Z. A. Shebli and B. D. Beheshti *A study on penetration testing process and tools* [cit. 2019-02-16]. DOI: 10.1109/LISAT.2018.8378035. ISBN 978-1-5386-5029-5. Dostupné z: https://ieeexplore.ieee.org/document/8378035

[7] *Penetration testing methodologies* [online]. [cit. 2019-02-16]. Dostupné z: https://www.owasp.org/index.php/Penetration_testing_methodologies

[8] *Open Source Security Testing Methodology Manual* [online]. [cit. 2019-02-16]. Dostupné z: http://www.isecom.org/research/

[9] SHOSTACK, Adam. *Threat Modeling: Designing for Security* Wiley, 2014. ISBN 978-1-118-82269-2.

[10] *Common Weakness Scoring System* [online]. [cit. 2019-02-16]. Dostupné z: https://cwe.mitre.org/cwss/cwss_v1.0.1.html

[11] *Common Weakness Enumeration* [online]. [cit. 2019-02-16]. Dostupné z: https://cwe.mitre.org/index.html

[12] *Common Vulnerability Scoring System* [online]. [cit. 2019-02-16]. Dostupné z: https://www.first.org/cvss/

[13] Study: A Penetration Testing Model. *Federal Office for Information Security (BSI)* [online]. [cit. 2019-02-17]. Dostupné z: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1

[14] KAKRANIA, Aditya. *Manual or Automated Application Security Testing: What's More Effective?* [online]. 13. 2. 2017 [cit. 2019-02-17]. Dostupné z: https://blog.securityinnovation.com/manual-or-automated-application-security-testing-whats-more-effective

[15] KUMAR, V.V.N. SURESH. Ethical Hacking and Penetration Testing Strategies. In: *International Journal of Emerging Technology in Computer Science and Electronics (IJETCSE)* [online]. s. 21-23 [cit. 2019-02-17]. ISSN 0976-1353. Dostupné z: http://www.ijetcse.com/wp-content/plugins/ijetcse/file/upload/docx/460ETHICAL-HACKING-AND-PENETRATION-TESTING-STRATAGIES-pdf.pdf

[16] *Dnsenum* [online]. [cit. 2019-02-20]. Dostupné z: https://github.com/fwaeytens/dnsenum/

[17] *Dnsrecon* [online]. [cit. 2019-02-20]. Dostupné z: https://github.com/darkoperator/dnsrecon/

[18] *Shodan.com* [online]. [cit. 2019-02-20]. Dostupné z: https://www.shodan.io

[19] *DNSdumpster.com* [online]. [cit. 2019-02-23]. Dostupné z: https://dnsdumpster.com/

[20] *The Harvester: E-mails, subdomains and names harvester* [online]. [cit. 2019-02-20]. Dostupné z: https://github.com/laramies/theHarvester

[21] *OpenVAS* [online]. [cit. 2019-02-20]. Dostupné z: http://www.openvas.org/

[22] *Nessus* [online]. [cit. 2019-02-20]. Dostupné z: https://www.tenable.com/products/nessus/nessus-professional

[23] *Nexpose* [online]. [cit. 2019-02-21]. Dostupné z: https://www.rapid7.com/products/nexpose/

[24] LEONOV, Alexander. Fast comparison of Nessus and OpenVAS knowledge bases. *Avleonov.com* [online]. [cit. 2019-02-21]. Dostupné z: https://avleonov.com/2016/11/27/fast-comparison-of-nessus-and-openvas-knowledge-bases/ https://github.com/Marten4n6/EvilOSX

[25] *W3af* [online]. [cit. 2019-02-21]. Dostupné z: https://github.com/andresriancho/w3af/

[26] *Nikto* [online]. [cit. 2019-02-21]. Dostupné z: https://github.com/sullo/nikto

[27] *WPScan* [online]. [cit. 2019-02-21]. Dostupné z: https://github.com/wpscanteam/wpscan

[28] *OWASP JoomScan* [online]. [cit. 2019-02-21]. Dostupné z: https://github.com/rezasp/joomscan

[29] *Metasploit - Penetration testing software for offensive security teams* [online]. [cit. 2019-02-23]. Dostupné z: https://www.rapid7.com/products/metasploit/

[30] *Immunity Canvas* [online]. [cit. 2019-02-23]. Dostupné z: https://www.immunityinc.com/products/canvas/

[31] *Secureauth Core Impact* [online]. [cit. 2019-02-23]. Dostupné z: https://www.secureauth.com/products/penetration-testing/core-impact

[32] *THC Hydra* [online]. [cit. 2019-02-23]. Dostupné z: https://github.com/vanhauser-thc/thc-hydra

[33] *Medusa* [online]. [cit. 2019-02-23]. Dostupné z: https://github.com/jmk-foofus/medusa

[34] *Ncrack* [online]. [cit. 2019-02-23]. Dostupné z: https://github.com/nmap/ncrack

[35] *Mitm Proxy* [online]. [cit. 2019-02-23]. Dostupné z: https://mitmproxy.org

[36] *Ettercap* [online]. [cit. 2019-02-23]. Dostupné z: https://github.com/Ettercap/ettercap

[37] *Bettercap* [online]. [cit. 2019-02-23]. Dostupné z: https://github.com/bettercap/bettercap

[38] *SQLMap* [online]. [cit. 2019-02-23]. Dostupné z: https://github.com/sqlmapproject/sqlmap

[39] *WordPress Exploit Framework* [online]. [cit. 2019-02-25]. Dostupné z: https://github.com/rastating/wordpress-exploit-framework

[40] *XAttacker* [online]. [cit. 2019-02-25]. Dostupné z: https://github.com/Moham3dRiahi/XAttacker

[41] *OWASP ZAP* [online]. [cit. 2019-02-25]. Dostupné z: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[42] *Pupy* [online]. [cit. 2019-02-27]. Dostupné z: https://github.com/n1nj4sec/pupy/

[43] *Powersploit* [online]. [cit. 2019-02-27]. Dostupné z: https://github.com/PowerShellMafia/PowerSploit

[44] *Powershell Empire* [online]. [cit. 2019-02-27]. Dostupné z: https://github.com/EmpireProject/Empire

[45] *How to bypass AMSI and execute ANY malicious Powershell code* [online]. [cit. 2019-02-27]. Dostupné z: https://0x00-0x00.github.io/research/2018/10/28/How-to-bypass-AMSI-and-Execute-ANY-malicious-powershell-code.html

[46] *Bella* [online]. [cit. 2019-02-27]. Dostupné z: https://github.com/kdaoudieh/Bella

[47] *EvilOSX* [online]. [cit. 2019-02-27]. Dostupné z: https://github.com/Marten4n6/EvilOSX

[48] *Hashcat* [online]. [cit. 2019-02-27]. Dostupné z: https://hashcat.net/hashcat/

[49] *John the Ripper* [online]. [cit. 2019-02-27]. Dostupné z: https://www.openwall.com/john/

[50] ERTAM, Fatih a Orhan YAMAN. *Intrusion detection in computer networks via machine learning algorithms* [online]. [cit. 2019-03-02]. DOI: 10.1109/IDAP.2017.8090165. Dostupné z: https://ieeexplore.ieee.org/document/8090165

[51] PARK, Kinam, Youngrok SONG a Yun-Gyung CHEONG. *Classification of Attack Types for Intrusion Detection Systems Using a Machine Learning Algorithm* [online]. [cit. 2019-03-02]. DOI: 10.1109/BigDataService.2018.00050. Dostupné z: https://ieeexplore.ieee.org/document/8405725

[52] MAKANDAR, Aziz a Anita PATROT. *Malware class recognition using image processing techniques* [online]. [cit. 2019-03-02]. DOI: 10.1109/ICDMAI.2017.8073489. Dostupné z: https://ieeexplore.ieee.org/document/8073489

[53] LUO, Jhu-Sin a Dan Chia-Tien LO. *Binary malware image classification using machine learning with local binary pattern* [online]. [cit. 2019-03-02]. DOI: 10.1109/BigData.2017.8258512. Dostupné z: https://ieeexplore.ieee.org/document/8258512

[54] OBES, Jorge Lucangeli, Carlos SARRAUTE a Gerardo RICHARTE. *Attack Planning in the Real World* [online]. 2013 [cit. 2019-03-02]. Dostupné z: https://arxiv.org/abs/1306.4044v2

[55] ALMUBAIRIK, Norah Ahmed a Gary WILLS. *Automated penetration testing based on a threat model* [online]. [cit. 2019-03-02]. DOI: 10.1109/ICITST.2016.7856742. Dostupné z: https://ieeexplore.ieee.org/document/7856742

[56] XUEQIU, Qiong JIA, Shuguang WANG, Chunhe XIA a Liangshuang LV. *Automatic Generation Algorithm of Penetration Graph in Penetration Testing* [online]. [cit. 2019-03-02]. DOI: 10.1109/3PGCIC.2014.104. Dostupné z: https://ieeexplore.ieee.org/document/7024641

[57] *GyoiThon* [online]. [cit. 2019-03-02]. Dostupné z: https://github.com/gyoisamurai/GyoiThon

[58] GHANEM, Mohamed C. a Thomas M. CHEN. *Reinforcement Learning for Intelligent Penetration Testing* [online]. [cit. 2019-03-02]. DOI: 10.1109/WorldS4.2018.8611595. Dostupné z: https://ieeexplore.ieee.org/document/8611595/

[59] *Deep Exploit* [online]. [cit. 2019-03-02]. Dostupné z: https://github.com/13o-bbr-bbq/machine_learning_security/tree/master/DeepExploit

[60] *Nmap* [online]. [cit. 2019-03-05]. Dostupné z: https://nmap.org/

[61] *Masscan* [online]. [cit. 2019-03-06]. Dostupné z: https://github.com/robertdavidgraham/masscan

[62] *Hping3* [online]. [cit. 2019-03-10]. Dostupné z: http://www.hping.org/

[63] DAVENDRA, Donald a Ivan ZELINKA. *Self-Organizing Migrating Algorithm: Methodology and Implementation.* Springer, Cham, 2016. ISBN 978-3-319-28161-2.

[64] ZELINKA I., OPLÁTKOVÁ Z., ŠEDÁ M., OŠMERA P., VČELAŘ F., Evolutionary techniques – principles and applications, BEN, Prague, 2008, 598 p.

[65] KENNEDY, J. a R. EBERHART. Particle swarm optimization. In: *Proceedings of ICNN'95 - International Conference on Neural Networks.* 1995, s. 1942-1948, vol. 4 [cit. 2019-03-12]. DOI: 10.1109/ICNN.1995.488968.

[66] ZELINKA, Ivan. *Biologicky inspirované výpočty: aneb vybrané statě z evolučních algoritmů.* Fakulta elektrotechniky a informatiky, VŠB TU Ostrava.

[67] MIRJALILI, Seyedali, Andrew LEWISA a Seyed Mohammad MIRJALILIB. Grey Wolf Optimizer. In: *Advances in Engineering Software: Volume 69* [online]. 2014, 2014, s. 46-61 [cit. 2019-03-12]. DOI: 10.1016/j.advengsoft.2013.12.007. ISSN ISSN 0965-9978. Dostupné z: https://www.sciencedirect.com/journal/advances-in-engineering-software/vol/69/suppl/C

[68] STORN, Rainer a Kenneth PRICE. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. In: *Journal of Global Optimization* [online]. Volume 11. 1997, 1997, s. 341-359 [cit. 2019-03-12]. DOI: 10.1023/A:1008202821328. Dostupné z: https://link.springer.com/article/10.1023%2FA%3A1008202821328

[69] Alexander CHOONG a Rami ZHU. *Parallelizing Simulated Annealing-Based Placement Using GPGPU* [online]. [cit. 2019-03-13]. DOI: 10.1109/FPL.2010.17. Dostupné z: https://ieeexplore.ieee.org/document/5694216

[70] SINGH, Vaibhav Kant a Shweta PANDEY. Minimum configuration MLP for solving XOR problem. In: *3rd International Conference on Computing for Sustainable Global Development (INDIACom)* [online]. New Delhi, India, 2016, 2016 [cit. 2019-03-13]. ISSN 0973 – 7529. e-ISBN: 978-9-3805-4421-2. Dostupné z: https://ieeexplore.ieee.org/document/7724250

[71] ZAKI, MOHAMMED J. a WAGNER MEIRA. *DATA MINING AND ANALYSIS: Fundamental Concepts and Algorithms.* Cambridge: Cambridge University Press, 2014. ISBN 978-0-521-76633-3.

[72] GAZALBA, Ikbal, Ikbal MUSTAKIM, REZA a OKFALISA. *Comparative analysis of k-nearest neighbor and modified k-nearest neighbor algorithm for data classification* [online]. 08 February 2018 [cit. 2019-03-13]. DOI: 10.1109/ICITISEE.2017.8285514. Dostupné z: `https://ieeexplore.ieee.org/document/8285514`

[73] SALZBERG, Steven L. *C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993* [online]. 1994, 235–240 [cit. 2019-03-13]. DOI: 10.1007/BF00993309. Dostupné z: `https://doi.org/10.1007/BF00993309`

[74] HO, Tin Kam. *Random decision forests* [online]. 1995 [cit. 2019-03-13]. DOI: 10.1109/ICDAR.1995.598994. Dostupné z: `https://ieeexplore.ieee.org/document/598994`

[75] FREUND, Yoav a Robert E. SCHAPIRE. *Experiments with a new boosting algorithm* [online]. 1996 [cit. 2019-03-13]. ISBN: 1-55860-419-7. Dostupné z: `http://dl.acm.org/citation.cfm?id=3091696.3091715`

[76] MOUSTAFA, Nour a Jill SLAY. *UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)* [online]. 10 December 2015 [cit. 2019-03-16]. DOI: 10.1109/MilCIS.2015.7348942. Dostupné z: https://ieeexplore.ieee.org/abstract/document/7348942

[77] *Argus* [online]. [cit. 2019-03-17]. Dostupné z: `https://qosient.com/argus/`

[78] *Scipy* [online]. [cit. 2019-03-18]. Dostupné z: `https://www.scipy.org/`

[79] *Scikit-learn* [online]. [cit. 2019-03-18]. Dostupné z: `https://scikit-learn.org/`

[80] *Redux-Saga* [online]. [cit. 2019-04-01]. Dostupné z: `https://redux-saga.js.org/`

[81] *Flux: Application architecture for building user interfaces* [online]. 2014 [cit. 2019-04-01]. Dostupné z: `https://facebook.github.io/flux/`

Seznam příloh

Příloha A: Analýza datové sady (2 strany)
Příloha B: Konfigurace testovacího prostředí (4 strany)

A Analýza datové sady

Tabulka 8: Výčet atributů datové sady - část 1/2

Číslo	Název	Typ	Popis
1	srcip	Nominální	Zdrojová IP
2	sport	Celočíselný	Zdrojový port
3	dstip	Nominální	Cílová IP
4	dsport	Celočíselný	Cílový port
5	proto	Nominální	Protokol
6	state	Nominální	Stav spojení
7	dur	Spojitý	Trvání spojení
8	sbytes	Celočíselný	Počet odeslaných bytů
9	dbytes	Celočíselný	Počet přijatých bytů
10	sttl	Celočíselný	Zdrojové TTL
11	dttl	Celočíselný	Cílové TTL
12	sloss	Celočíselný	Počet ztracených odeslaných paketů zdrojem
13	dloss	Celočíselný	Počet ztracených odeslaných paketů cílem
14	service	Nominální	Služba
15	Sload	Spojitý	Počet odeslaných bitů za sekundu zdrojem
16	Dload	Spojitý	Počet odeslaných bitů za sekundu cílem
17	Spkts	Celočíselný	Počet odeslaných paketů za sekundu zdrojem
18	Dpkts	Celočíselný	Počet odeslaných paketů za sekundu cílem
19	swin	Celočíselný	Velikost Sliding Window zdroje
20	dwin	Celočíselný	Velikost Sliding Window cíle
21	stcpb	Celočíselný	TCP sekvenční číslo zdroje
22	dtcpb	Celočíselný	TCP sekvenční číslo cíle
23	smeansz	Celočíselný	Průměrný počet odeslaných paketů zdrojem
24	dmeansz	Celočíselný	Průměrný počet odeslaných paketů cílem
25	trans depth	Celočíselný	Hloubka HTTP transakce

Tabulka 9: Výčet atributů datové sady - část 2/2

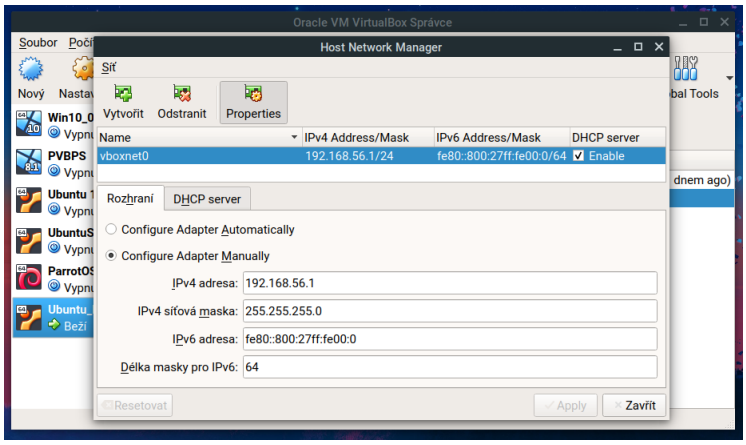
Číslo	Název	Typ	Popis
26	res bdy len	Celočíselný	Velikost body u HTTP
27	Sjit	Spojitý	Jitter zdroje
28	Djit	Spojitý	Jitter cíle
29	Stime	Časové razítko	Počátek spojení
30	Ltime	Časové razítko	Konec spojení
31	Sintpkt	Spojitý	Inter-arrival čas paketů zdroje
32	Dintpkt	Spojitý	Inter-arrival čas paketů cíle
33	tcprtt	Spojitý	RTT u TCP
34	synack	Spojitý	Čas mezi SYN a SYN ACK
35	ackdat	Spojitý	Čas mezi SYN ACK a ACK
36	is sm ips ports	Binární	1 pokud jsou stejné IP/port zdroje a cíle
37	ct state ttl	Celočíselný	Hodnota kombinující STTL/DTTL a stav
38	ct flw http mthd	Celočíselný	Počet GET/POST požadavků u HTTP
39	is ftp login	Binární	1 pokud došlo k přihlašení k FTP
40	ct ftp cmd	Celočíselný	Počet FTP příkazů
41	ct srv src	Celočíselný	Počet spojení ze stejné IP a portu ze 100 spojení
42	ct srv dst	Celočíselný	Počet spojení k stejné IP a portu ze 100 spojení
43	ct dst ltm	Celočíselný	Počet spojení k stejné IP ze 100 spojení
44	ct src ltm	Celočíselný	Počet spojení ze stejné IP ze 100 spojení
45	ct src dport ltm	Celočíselný	Počet spojení se stejným zdr./cílovým portem
46	ct dst sport ltm	Celočíselný	Počet spojení ke stejné IP a portu
47	ct dst src ltm	Celočíselný	Počet spojení ke stejné IP z stejné IP
48	attack cat	Nominální	Kategorie provozu
49	Label	Binární	1 pro útok, 0 pro běžný provoz

B Konfigurace testovacího prostředí

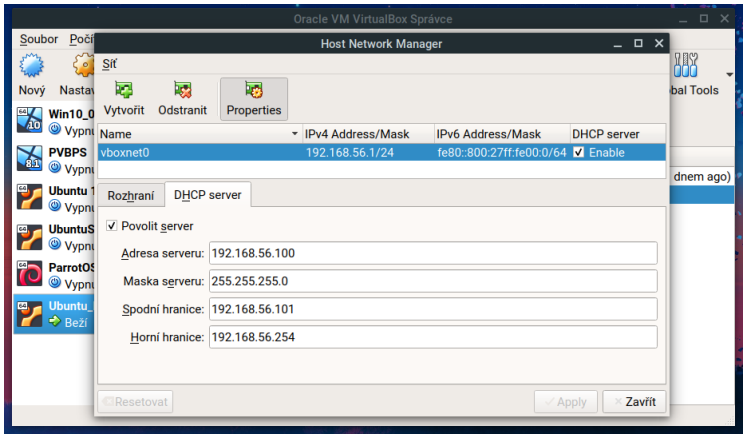
V kapitole 6 je stručně popsáno navržené testovací prostředí, které se skládá ze dvou virtuálních strojů, a to PentestVM a IdsVM. Oba virtuální stroje používají operační systém Ubuntu Server 18.04.2 LTS. Následující postup je určen pro linuxové distribuce založené na Debianu, jako je právě zmíněné Ubuntu, nicméně princip je velmi podobný i pro další distribuce.

B.1 Základní konfigurace síťového adaptérů

Oba virtuální stroje používají virtualizační software VirtualBox, vzorová konfigurace používá tzv. host-only síťový adaptér se sítí 192.168.56.1/24. PentestVM má přiřazenou IP adresu 192.168.56.21 a IdsVM je dostupné na 192.168.56.10. Ukázku konfigurace vidíme na Obrázcích 66 a 67.



Obrázek 66: Konfigurace síťového adaptéru část 1/2



Obrázek 67: Konfigurace síťového adaptéru část 2/2

B.2 Konfigurace virtuálního stroje IdsVM

Prvním krokem je přiřazení IP adresy host-only adaptéru. V ukázkové konfiguraci je pojmenován jako `enp0s8`, viz Výpis 16. Druhým krokem je přenesení obsahu složky `/Deploy/IDS`, která je součástí elektronické přílohy, na server, např. pomocí nástroje sftp. Dalším krokem je zprovoznění síťových služeb ve formě SNMP, DNS a HTTP serveru. K tomuto účelu je k dispozici script `install.sh`, který stačí pouze spustit s právy uživatele `root` a vše potřebné bude nainstalováno. Posledním krokem je instalace závislostí samotného IDS, k tomuto účelu je opět k dispozici script `setup_python.sh`. Po instalaci všech závislostí už stačí jen spustit nástroj Argus a následně samotné IDS. Pro spuštění Argusu je k dispozici script `start_argus.sh`, je potřeba pouze zajistit, aby Argus běžel s právy roota. Před spuštěním IDS je potřeba aktivovat virtuální prostředí, poté stačí spustit script `ids_app.py`, jehož parametrem je cesta je konfiguračnímu souboru.

```
# Přiřazení IP adresy
sudo ip addr add dev enp0s8 192.168.56.10/24

# Instalace síťových služeb
sudo ./install.sh

# Instalace závislostí IDS
./setup_python.sh

# Spuštění nástroje Argus
sudo argus -P 51000 -i enp0s8

# Spuštění IDS
source venv/bin/activate
python3 ids_app.py learn.conf
```

Výpis 16: Konfigurace virt. stroje IdsVM

Pro konfiguraci IDS slouží konfigurační soubor, jehož cesta představuje jediný parametr `ids_app.py`. Ukázkový konfigurační soubor společně s vysvětlením parametrů vidíme ve Výpisu 17.

```
[IDS]
log_file = ./resources/ids.log # Soubor pro logování
# Soubor s modelem, který bude načten - POZOR pro načtení je nutno nastavit
    parametr learn_malicious_traffic na False
model_path = ./resources/models/rf.sav
# CSV soubor s provozem, který je považován za neškodný
base_data_path = ./resources/data/normal_syntetic_smote.csv
argus_ip = localhost # IP adresa Argus server
argus_port = 51000 # Port Argus serveru
malicious_ip = 192.168.56.21 # IP adresa generátoru provozu, který je považován
     za útok
learn_malicious_traffic = True # Hodnota True zapíná učící režim
gen_report = False # Generování reportu - vhodné pro testovací režim
save_model_path = ./resources/models/model_01.sav # Cesta, kam bude uložen nový
     model
report_path = report_learn_01.csv # Cesta, kam bude uložen případný report

# REST API endpoint, kam bude odesílána zpětná vazba
api_url = http://192.168.56.21:80/api/v1/ids/probability
```

Výpis 17: Konfigurační soubor IDS

B.3 Konfigurace virtuálního stroje PentestVM

Prvním krokem je opět přiřazení IP adresy, tedy 192.168.56.21 na host-only adaptér, aby mohl virtuální stroj komunikovat s dalšími virtuálními stroji. Dále následuje instalace závislostí, v podobě Java 10 a Python knihoven. Opět jsou k dispozici scripty `install.sh` a `setup_python.sh`. Poté už následuje spuštění webové aplikace, příkazem `java -jar pentest-web-app-0.0.1-SNAPSHOT`. Posledním krokem je spuštění generátoru provozu. Generátor provozu je nutné spouštět s právy roota, protože využívá nízko-úrovňový přístup k socketům.

```
# Přiřazení IP adresy
sudo ip addr add dev enp0s8 192.168.56.21/24

# Instalace Javy
sudo ./install.sh

# Instalace závislostí generátoru provozu
./setup_python.sh

# Spuštění webové aplikace
java -jar pentest-web-app-0.0.1-SNAPSHOT

# Spuštění generátoru provozu
source venv/bin/activate
sudo ./main.py
```

Výpis 18: Konfigurace virt. stroje PentestVM